PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

# ALGORITMOS DE CONSISTENCIA PARA PROGRAMACIÓN CON RESTRICCIONES

**SEBASTIÁN ANDRÉS LIZAMA AVILÉS**
**ALEXIS IVÁN MUÑOZ CONCHA**

INFORME FINAL DE PROYECTO
PARA OPTAR AL TÍTULO PROFESIONAL DE
INGENIERO DE EJECUCIÓN EN INFORMÁTICA

ABRIL, 2013

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

# ALGORITMOS DE CONSISTENCIA PARA PROGRAMACIÓN CON RESTRICCIONES

**SEBASTIÁN ANDRÉS LIZAMA AVILÉS**
**ALEXIS IVÁN MUÑOZ CONCHA**

**Ricardo Soto de Giorgis**
**Profesor Guía**

**Broderick Crawford Labrín**
**Profesor Co-referente**

ABRIL, 2013

## *Dedicatoria*

*A nuestras familias y amigos, que siempre nos han apoyado.*

## *Agradecimientos*

*A nuestras familias, que siempre nos acompañaron y apoyaron en este largo proceso universitario, entregándonos las herramientas necesarias para cumplir con éxito esta etapa de nuestras vidas.*

*A nuestros profesores, que a lo largo de la carrera fortalecieron nuestras virtudes y conocimientos, logrando así ser unos alumnos preparados para afrontar el difícil mundo laboral que nos espera. Especialmente a nuestro profesor guía, el que nos apoyó y comprendió en todo momento y que siempre estuvo cuando lo necesitamos.*

*A nuestros compañeros y amigos de siempre, que nos demostraron en todo momento su cariño y apoyo incondicional.*

## Resumen

La *Programación con Restricciones* es un paradigma de programación dedicado a la resolución eficiente de *problemas de satisfacción de restricciones*, comúnmente conocidos como *CSP*. Un CSP se compone de una secuencia de variables, cada una asociada a un dominio y un conjunto de restricciones sobre esas variables. Para la resolución de un CSP se utilizan variados tipos de algoritmos, uno de los más utilizados son los algoritmos de búsqueda completa. Estos comúnmente se combinan con un proceso de filtraje que permite eliminar valores de los dominios que no conducen a ninguna solución. Este proceso de filtraje se realiza por medio de técnicas de consistencia, siendo la *Arco-consistencia* una de las más utilizadas. El presente proyecto se centra en integrar un algoritmo de filtraje en un solver experimental llamado *RS-Solver*. En particular, se implementará una versión modificada del *algoritmo AC-1* que permite reducir un CSP dado a su correspondiente mínimo equivalente. Esto como consecuencia permitirá mejorar el rendimiento actual del RSSolver.



**Palabras Clave:**Programación con Restricciones, CSP, Arco-consistencia, AC-1, RSSolver.

## Abstract

*Constraint Programming* is a programming paradigm devoted to the efficient solving of *constraint satisfaction problems (CSP)*. A CSP is composed of a sequence of variables, each one associated to a domain and a set of constraints over these variables. For the CSP solving, various algorithms can be employed, one of the most used are the complete search ones. These algorithms are commonly combined with a filtering process that allows one to delete those values from domains that do not lead to any solution. This filtering process is performed by means of consistency techniques, the *Arc-consistency* being one of the most used. The present project focuses on the integration of a filtering algorithm in an experimental solver called *RSSolver*. In particular, a modified version of the *AC-1 algorithm* will be implemented, that allows one to reduce a given CSP to its corresponding minimum equivalent. As a consequence, the performance of the current RSSolver will be improved.



**Keywords:**Constraint Programming, CSP, Arc Consistency, AC-1, RSSolver.

# Índice

# Lista de Figuras

## Lista de Tablas

# 1. Introducción

En informática, la *Programación con Restricciones* o *CP* (en inglés *Constraint Programming*) es un paradigma, que se encarga de darle solución a *problemas de satisfacción de restricciones* o *CSP* (en inglés *Constraint Satisfaction Problem*). Un CSP se compone de una secuencia de variables, cada una asociada a un dominio y un conjunto de restricciones sobre esas variables. Para la resolución de un CSP se utilizan variados tipos de algoritmos, uno de los más utilizados son los algoritmos de búsqueda completa, los cuales se encargan de recorrer todas las potenciales soluciones que se pueden generar de un problema. Un ejemplo conocido de búsqueda completa es el realizado por el clásico algoritmo *Backtracking*.

El algoritmo Backtracking ha sido ampliamente utilizado para resolver CSPs. Sin embargo, el Backtracking puro puede resultar demasiado costoso cuando el problema posee un número significativo de variables. Por lo mismo, dentro de la programación con restricciones, el Backtracking se suele combinar con un proceso de filtraje. La idea es eliminar valores de los dominios que no conduzcan a ninguna solución. Este proceso de filtraje se llama técnicamente propagación de restricciones y se realiza por medio de técnicas de consistencia. Una de la técnicas de consistencia más conocidas es la *arco-consistencia* de la cual existen diversas variantes tales como AC-1, AC-2, AC-3 y AC4 entre otras. El presente proyecto se centra en la implementación de un algoritmo de arco-consistencia para el *RSSolver* (solver implementado por el Dr. Ricardo Soto) con el fin de complementar el algoritmo de Backtracking ya implementado. En particular se implementará una versión modificada del algoritmo AC-1 que permite reducir un CSP dado a su correspondiente mínimo equivalente. Esto como consecuencia permitirá mejorar el rendimiento actual del RSSolver.

El presente documento se organiza de la siguiente manera. En el capítulo 2, se describen los objetivos del proyecto. En el capítulo 3, se presenta el estado del arte, donde se describen las distintas variantes investigadas de la arco-consistencia. En el capítulo 4, se describen las bases de la programación con restricciones y los algoritmos clásicos de búsqueda completa. En el capítulo 5, se presentan detalladamente las versiones más conocidas de la arco-consistencia. En el capítulo 6, se presenta teóricamente la parte inicial del funcionamiento del solver (ANTLR). El RSSolver se describe en el capítulo 7, seguido de la implementación realizada. Una breve experimentación se realiza en el capítulo 8, para finalizar con las conclusiones.

# 2. Definición de Objetivos

## 2.1. Objetivo General

- Implementar un algoritmo que permita reducir un CSP a su mínimo equivalente por medio de arco-consistencia.

## 2.2. Objetivos Específicos

- Comprender los diferentes algoritmos de Arco Consistencia.
- Comprender el algoritmo de Backtracking y Forward Checking.
- Implementar un algoritmo de arco consistencia que entregue como resultado un CSP mínimo equivalente.

# 3. Estado del Arte

La programación con restricciones es una tecnología emergente utilizada en diversos ámbitos para el estudio y posterior resolución de problemas basados en restricciones. Cada una de estas restricciones va a ser satisfecha con elementos de los dominios de las variables asociadas a ésta, de la misma forma, cuando cada una de las restricciones asociadas al problema sea complacida, se estará en presencia de una solución.

A grandes rasgos no parece muy complejo, pero si se toma en cuenta un problema pequeño, sin considerar la cantidad de restricciones, que contenga 7 variables con un promedio de 10 valores por dominio, se tendría una cantidad mínima aproximada de 283 millones de posibilidades, teniendo en cuenta a su vez que puede que sólo una de estas sea la solución. Debido a esta complejidad demostrada de forma simple anteriormente, se hace necesario utilizar de forma adecuada las técnicas disponibles para la solución de los problemas, considerando técnicas de búsqueda y técnicas de inferencia, además de ciertas heurísticas, las cuales van a permitir ordenar, ya sea las variables, valores y/o restricciones, logrando resolver los problemas de forma más eficiente.

## 3.1. Generalidades

Existen ciertos conceptos asociados a las matemáticas, los cuales ya han sido nombrados anteriormente, como dominio, variable o restricción. Para comprender a fondo la investigación se hace necesario tomar en cuenta ciertas definiciones, que son específicas del tipo de problemas a investigar, algunas de estas son:

- En el caso de las restricciones existen tres tipos: unarias, binarias y no-binarias, considerando una, dos o muchas variables asociadas respectivamente.
- Las restricciones serán normalizadas, cuando una restricción asociadas a un problema no involucra exactamente las mismas variables de una restricción anterior o posterior, de lo contrario serán consideradas no-normalizadas.
- Un problema será considerado discreto cuando posea variables con dominios finito y continuo cuando posea variables con dominios continuos.
- Algoritmos de búsqueda, son aquellos algoritmos que permiten recorrer todas las posibles soluciones a un problema, hasta encontrar una

solución.

- Técnicas de consistencia, permiten mejorar la eficiencia de los algoritmos de búsqueda, y a raiz de ésto, reducir los nodos a instanciar en el árbol de búsqueda.
- Arco-consistencia, utilizada en restricciones binarias o también denominados arcos, busca que cada valor del dominio de una variable sea soportado por algún elemento del dominio de la variable restante asociada a la restricción, eliminando los valores que no serán parte de una solución.

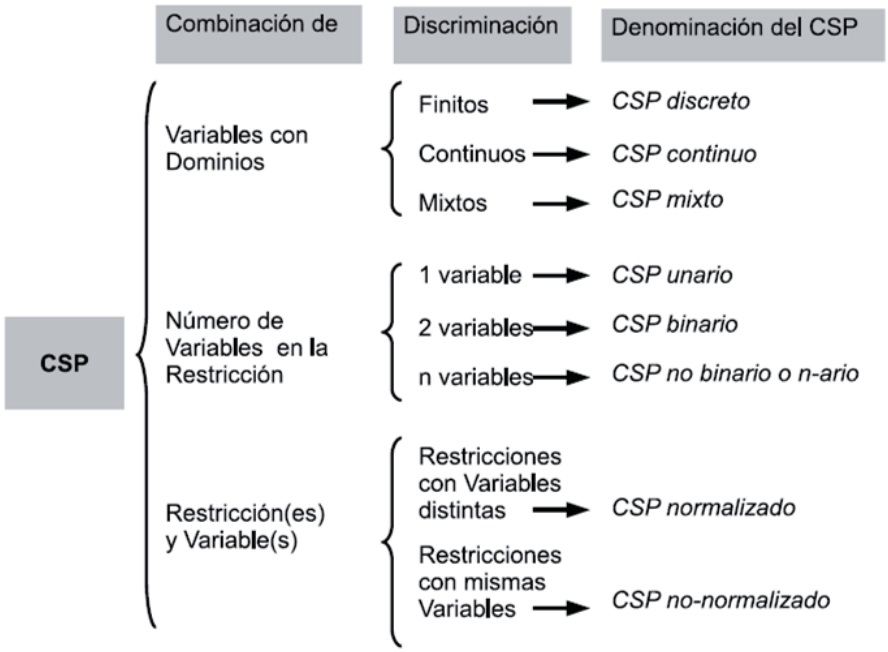



Figura 1: Tipos de CSP según características que posean.

Teniendo claros estos conceptos(ver Figura 1), es bueno mencionar que en general, las investigaciones asociadas a la programación con restricciones son realizadas sobre problemas binarios normalizados, debido a que es más fácil modelarlos, simplificando de esta forma la explicación de los algoritmos propuestos. No obstante, existen investigaciones actuales que buscan mejorar

el funcionamiento de los algoritmos de arco-consistencia más básicos, es decir, desde AC-1 hasta AC-4(ver Capítulo 5), propuestos entre 1970 y 1986.

## 3.2. Investigaciones Previas

Sabiendo que todo aquello que está propuesto es posible mejorarlo, se han creado en la actualidad un número no menor de algoritmos de arco-consistencia, los cuales eliminan ciertas limitantes asociadas a la funcionalidad de estos, logrando mejorar distintos aspectos o características de los algoritmos, obteniendo de esta forma otros nuevos. Algunas de las investigaciones más actuales se detallan a continuación.

- AC3-OP [3]: Mejora el algoritmo de arco-consistencia AC3, logrando reducir el número de propagaciones hasta en un 50 % y con ello el tiempo y el número de chequeos de restricciones.
- AC3-NN [4]: Es una reformulación del algoritmo AC3, para procesar CSPs binarios con restricciones no-normalizadas y realizar correctamente la propagación de arcos que comparten las mismas variables.
- AC4-OP [2]: Poda el mismo espacio de búsqueda que AC4, pero mejora su eficiencia tanto en la inicialización (que la realiza bidireccionalmente, ya que sólo chequea las restricciones binarias normalizadas en un sentido) como en la propagación (sólo con variables que sean soporte de otras). AC4-OP es capaz de reducir el número de chequeos de restricciones en un 50 % y el número de propagaciones en un 15 %.
- AC4-OPNN [3, 1]: Poda el mismo espacio de búsqueda que AC4 y AC4-OP, pero mejora su eficiencia tanto en la inicialización que la realiza bidireccionalmente (sólo chequea las restricciones binarias no-normalizadas en un sentido) como en la propagación (sólo con variables que sean soporte de otras). Al igual que AC4-OP, AC4-OPNN es capaz de reducir el número de chequeo de restricciones en un 50 % y el número de propagaciones en un 15 % en CSPs no-normalizados.

Los algoritmos previamente resumidos buscan que las investigaciones previas cambien su enfoque o algunas caracterizas de ellos, pasando por ejemplo de resolver restricciones normalizadas a no normalizadas, ampliando de esta forma las posibilidades en el modelamiento de los problemas. La investigación actual se basa en el estudio de los algoritmos de arco-consistencia, queriendo enfocarse en un filtrado para el algoritmo de búsqueda Backtracking desarrollado en un prototipo de solver. Realizando un estudio sobre los algoritmos

propuestos entre 1970 y 1986, considerando algunos de los beneficios de ellos, se busca modificar el algoritmo AC1, para crear un filtro capaz de superarlo.

# 4. Programación con Restricciones

En el año 1963 ya se hacía alusión al uso de restricciones por medio del estudio realizado por Ivan Sutherland para su tesis doctoral *Sketchpad: A Man-Machine Graphical Communications System*, lo cual fue un gran avance en la informática, ya que se creó el primer sistema de dibujo basado en un lápiz y una pantalla, para realizar esta tarea se hacía necesario el uso de restricciones. Ya hacia los años 80' el estudio de CP (Constraint Programming) se hizo más fuerte, donde se llegó a utilizar lenguajes para el desarrollo de CP, como LISP. Llegando a fines de ésta década donde CP se utilizaba como extensión de la programación lógica, nació el concepto Constraint Logic Programming o con sus siglas CLP. A medida que el tiempo transcurría los estudios seguían avanzando, con esto muchas nuevas aplicaciones fueron desarrolladas, entre las que se destacan fuertemente los campos de la Investigación Operativa y Análisis Numéricos.

Tomando en cuenta parte de la historia hoy es posible mencionar que CP es aplicable en diversas áreas llegando incluso a mezclarlo con problemas de la vida diaria, que pueden ser expresados en base a restricciones, para lo cual se necesita definir ciertas variables las que al estar instanciadas en su totalidad deben satisfacer cada una de las restricciones planteadas, si esto se logra estamos en presencia de una solución (no necesariamente óptima). Todo dilema que presente un desarrollo como el antes mencionado, se le conoce como problema de satisfacción de restricciones o CSP.

## 4.1. Problemas de satisfacción de restricciones

Existe un sinnúmero de problemas asociados a distintas áreas (inteligencia artificial, investigación operativa, sistemas de recuperación de información) y diversas aplicaciones (planificación, diseño en la ingeniería, diagnóstico), los que a través del paradigma de la programación con restricciones, pueden ser modelados como problemas de satisfacción de restricciones o CSP. Básicamente un CSP se puede describir como un proceso en el que se evalúa un conflicto, el cual posee una serie de limitantes, sujetas a un dominio específico asociado al problema.

A la hora de llevar a la práctica este proceso para la resolución de algún CSP, se requiere abordar una primera fase, la que se denomina fase de modelado, en esta parte del proceso, se requiere representar el problema utilizando los elementos necesarios de la sintaxis utilizada para un CSP: variables dominios y restricciones.

Una *variable* es un símbolo que representa un elemento de un conjunto

dado, este conjunto, también llamado *dominio*, se muestra como una cantidad finita de valores posibles para una variable, finalmente las restricciones son las relaciones entre estas variables. Una solución se logra cuando se presenta un escenario en que cada variable posee un valor dentro de un dominio dado, y que a su vez cumple con satisfacer cada una de las restricciones asociadas al problema. Las restricciones pueden ser representadas de diversas formas dentro de un problema, ya sea por ecuaciones, o por medio un conjunto de tuplas válidas y no válidas para el problema. De la misma forma, de acuerdo a la cantidad de variables que poseen las restricciones es posible hablar de restricciones unarias (una variable), binarias (dos variables) y no-binarias (tres o más variables). Es muy importante tomar en cuenta estos aspectos al momento de realizar la fase de modelado, ya que las restricciones son determinantes al momento de evaluar la eficiencia de la resolución.

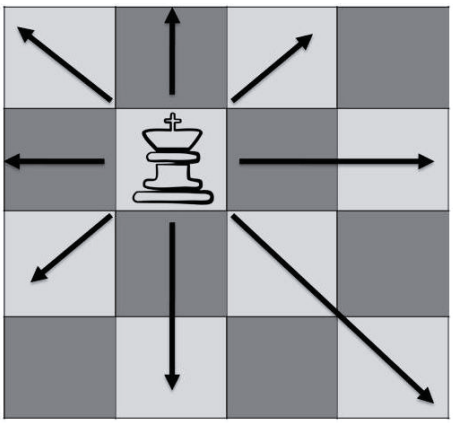


Figura 2: Posibles movimientos de una reina en el ajedrez.

Cada uno de los elementos básicos asociados a la primera fase (modelado) de un CSP será ilustrado, para un mejor entendimiento, mediante el clásico problema de las n-reinas, el cual consiste en ubicar en un tablero de ajedrez de $n \times n$, $n$ reinas, de tal forma que cada una de ellas, no sea una amenaza para las $n-1$ reinas restantes. Cabe mencionar que dentro de las reglas del ajedrez, está especificado que la reina amenaza a cada una de las figuras que estén posicionadas en su misma vertical, horizontal y además en sus diagonales. Estos movimientos serán los que determinen las restricciones del problema, movimientos que son representados en la Figura 2.

Una de las posibilidades para el modelado del problema, indica que cada

posición de la reina en la fila del tablero debe ser representada como variable y cada posición dentro de la columna como elementos del dominio del problema. De esta forma se tiene que para cada variable $X_i$ existe un valor $a$ asociado perteneciente al dominio, que a su vez significa que hay una reina $X$ en la intersección de la fila $i$ con la columna $a$, con $1 \leq a \leq n$ y $1 \leq i \leq n$.

Finalmente, existe la incógnita de cómo representar que cada reina debe encontrarse en una posición de tal forma que no amenace a ninguna de las restantes, esto se logra mediante la siguiente restricción. Para ello se toman dos reinas $X_i$ y $X_j$.

$$R_{ij} = \{(a, b)|a \neq b \wedge |i - j| \neq |a - b|\}\,(i > j)$$

Esto quiere decir que para $i$ y $j$ se tomarán los valores $a$ y $b$, de tal manera que $a$ sea distinto de $b$ (restricción para las columnas), por otra parte $i - j$ debe ser distinto que $a - b$ tomando en cuenta los valores absolutos de los resultados (restricción para las diagonales), finalmente $i$ debe ser menor que $j$ para considerar esta restricción. Tomando en cuenta cada una de las partes, el modelo del problema considerando 4 reinas queda expresado de la siguiente forma:

Variables: $X = \{X_1, X_2, X_3, X_4\}$, una variable por cada reina.
Dominios: $D_i = \{1, 2, 3, 4\}$, un valor por cada columna del tablero.
Restricciones: $\forall X_i, X_j$ con $1 \leq i, j \leq 4$ y $i \leq j$.

$$R_{ij} = \{(a, b)|a \neq b \wedge |i - j| \neq |a - b|\}\,(i > j)$$

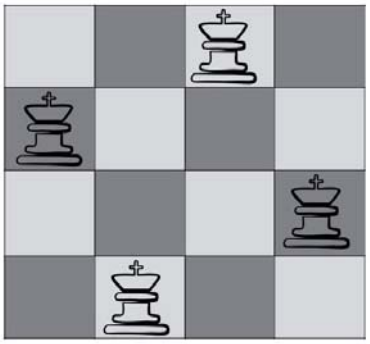


Figura 3: Solución al problema de las 4-reinas.

Tras la etapa de modelado o modelamiento se debe continuar con la segunda fase, la que se denomina fase de solución en donde se deben aplicar las técnicas para la satisfacción de restricciones, llegando así a la resolución del *CSP* (Figura 3). En gran parte del proceso afecta la selección del algoritmo de búsqueda, éste nos indica de qué forma se realiza el recorrido del árbol de soluciones para llegar a dicho resultado.

## 4.2. Algoritmos de Búsqueda

Existen diferentes técnicas o algoritmos para recorrer las posibles soluciones o estados de un problema de satisfacción de restricciones, viéndolo de forma general existen dos grandes grupos para realizar la búsqueda, los algoritmos completos y los incompletos.

Los algoritmos denominados incompletos o locales no garantizan una solución o respuesta óptima, debido a que la búsqueda se realiza sobre una sección de la totalidad del espacio de estados. A raíz de que no se realiza una evaluación minuciosa de las soluciones y gracias a la incorporación de algunas heurísticas en el proceso, estos algoritmos realizan la búsqueda con mayor rapidez. Teniendo una solución inicial, se itera repetidas veces dirigiéndose hacia otras soluciones que intentan mejorar el valor de la función objetivo y/o reducir el número de inconsistencias. En la mayoría de los casos, estos algoritmos finalizan tras una serie de intentos o iteraciones, o cuando encuentran el óptimo (si son capaces de detectarlo) [19].

Los algoritmos completos o sistemáticos son aquellos que a través de un árbol buscan soluciones asignándole posibles valores a las variables, si no existe una solución o no es posible dar una respuesta, esto es demostrado. El árbol de búsqueda representa cada una de las posibles asignaciones realizables, ya sea de la totalidad de las variables (nodos hojas), o de una parte de ellas (nodos intermedios), además el nodo raíz es aquel que representa el problema sin variables instanciadas. El algoritmo de búsqueda llamado Backtracking (BT), es considerado la base de la búsqueda sistemática, la gran diferencia con los algoritmos incompletos recae en la capacidad de encontrar y garantizar una solución, tomando en cuenta la excepción ya mencionada, el particular caso de que no exista solución. En este informe serán presentados los siguientes algoritmos sistemáticos: Generate and test, Backtracking, Forward checking y Mantenimiento de la Arco-consistencia.

### 4.2.1. Generate and Test

Este algoritmo es el más sencillo en cuanto a su funcionamiento, ya que recorre la totalidad del árbol, componiendo de forma metódica cada una de las asignaciones completas posibles. Cuando llega al término de una rama, es decir a un nodo hoja, teniendo todas las variables instanciadas, comprueba si se satisfacen todas las restricciones. Es en esta última frase en donde se hace notar la inconsistencia del algoritmo, ya que realiza asignaciones innecesarias generando estados en que es imposible llegar a una solución.

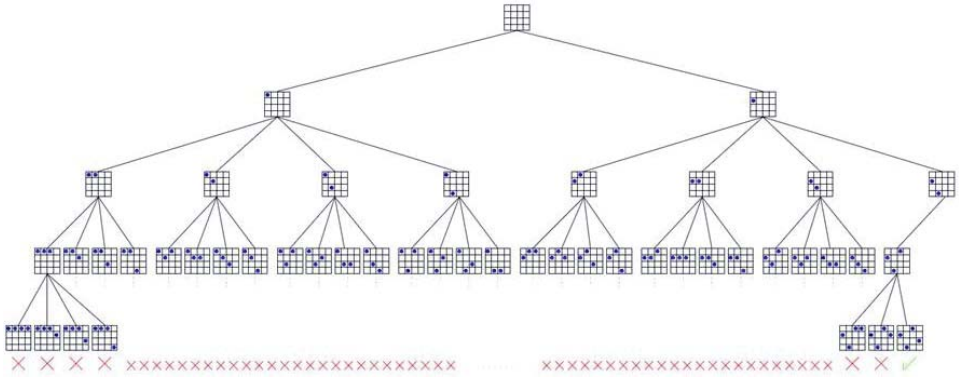


Figura 4: Proceso completo utilizando generate and test, para el problema de 4-reinas.

### 4.2.2. Backtracking

Backtracking trabaja de la misma forma que lo hace el algoritmo Generate and Test, pero con algunas diferencias, mediante las que se logra suplir la inconsistencia de este último. Se continúa realizando una búsqueda en profundidad por cada nodo, pero cada vez que se asigna un valor a la variable con la que actualmente se trabaja la búsqueda ($X_i$), se realiza una verificación del escenario o solución parcial (sin la totalidad de las variables instanciadas), de tal forma que, si el valor asignado a la variable $X_i$ produce una inconsistencia, considerando las variables anteriormente asignadas, el algoritmo retrocede y vuelve a asignar un nuevo valor del dominio a la misma variable ($X_i$),debido a que esta asignación parcial no formará parte de ninguna solución, ahorrándose de esta forma el recorrido del árbol que cuelga de la asignación parcial, este proceso se realiza hasta agotar cada una de las posibilidades del dominio ($D_i$) de la variable actual. Si se agotan las

posibilidades, BT retrocede nuevamente, pero cambia el valor de la variable anterior $X_{i-1}$. De esta forma se continúa con el funcionamiento del algoritmo hasta encontrar una solución, esto se logra cuando cada una de las asignaciones *actuales* realizadas en conjunto con las asignaciones pasadas (si es que hay), concuerdan sistemáticamente con las restricciones del problema. Si se agotan todas las posibilidades, queda demostrado que no existe una solución para el problema en cuestión.

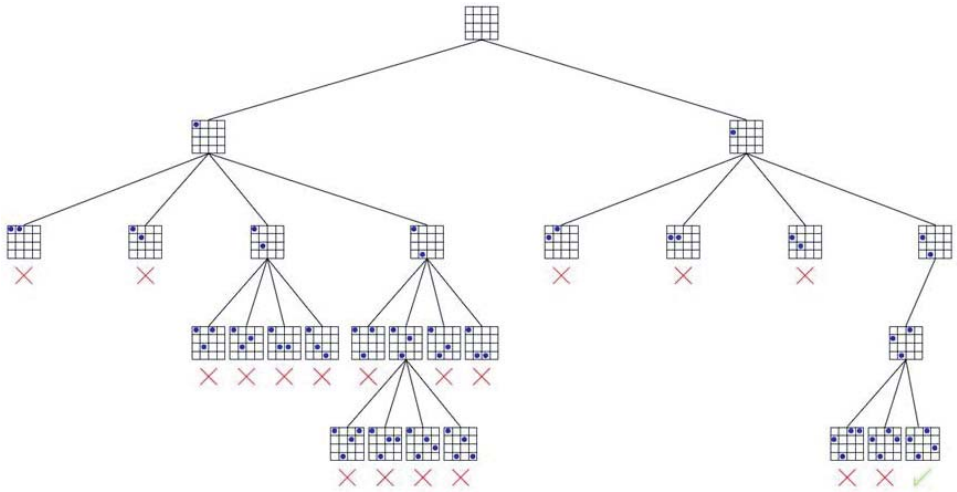


Figura 5: Proceso completo utilizando backtracking, para el problema de 4-reinas.

### 4.2.3. Forward checking

El algoritmo *forward checking* posee un enfoque Look-Ahead, esto quiere decir que realiza un chequeo hacia adelante en cada una de las etapas de la búsqueda. El FC es el algoritmo más común en este enfoque, logrando garantizar que cada una de las soluciones parciales sea consistente con cada valor de las variables futuras, esto se logra mediante la eliminación parcial de los valores inconsistentes en los dominios de las variables futuras, considerando como base la variable actual, es decir utiliza técnicas de consistencia. En el caso de que una variable futura ($X_{i+1}$) se quede sin dominio ($D_{i+1}$), debido a que todos sus valores han sido temporalmente eliminados, se instancia la variable actual ($X_i$) con un nuevo valor del dominio ($D_i$). De esta forma si ningún valor es consistente, se realiza un backtrack.

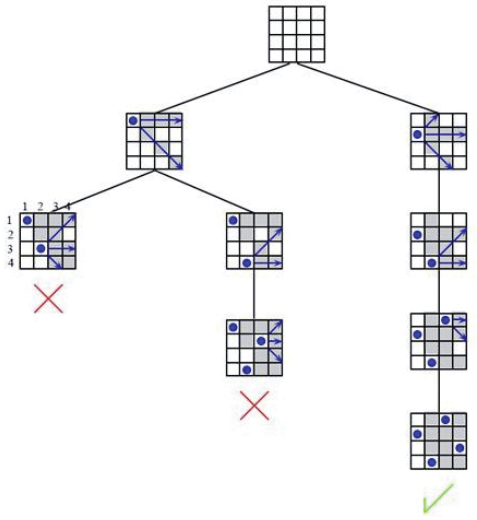


Figura 6: Proceso completo utilizando forward checking, para el problema de 4-reinas.

#### 4.2.4. Mantenimiento de la Arco-consistencia

Este tipo de algoritmos de búsqueda, que anteriormente se conocían como Real-Full-Look-Ahead (RFLA), son los más utilizados para la resolución de los CSP. Lo que se hace es combinar alguna técnica de inferencia incompleta (arco-consistencia), con un algoritmo de búsqueda, intentando de esta forma reducir el espacio de búsqueda de una manera más temprana que en el caso del algoritmo FC.

### 4.3. Técnicas de Consistencia

Para la resolución de un CSP se hace necesaria la utilización de ciertas técnicas que puedan mejorar el funcionamiento de los algoritmos de búsqueda, ya que estos, por si solos, son insuficientes a la hora de resolver un problema de buena manera. Si se considera el siguiente modelo:

**Variables**:$\{X, Y, Z\}$
**Dominios**: $D_x = \{0, 1\}$, $D_y = \{2, 3\}$, $D_z = \{1, 2\}$
**Restricciones**:$R_1 = X < Z$, $R_2 = X \neq Y$

En este caso no hay respuesta, si se pudiera identificar la inconsisten-

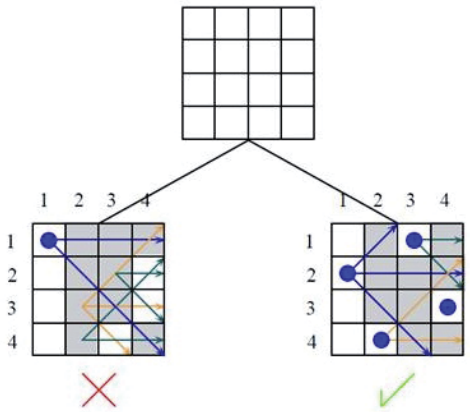



Figura 7: Proceso completo utilizando MAC, para el problema de 4-reinas.

cia con anterioridad, se evitaría el esfuerzo utilizado en la búsqueda de la solución, que finalmente no existe, es decir, que es posible identificar las inconsistencias antes del proceso de búsqueda y no necesariamente en ella. Es por esto que se encuentran ciertos métodos que ayudan a que la búsqueda se realice de forma más eficiente, reduciendo aquellos problemas que obstaculizan el buen funcionamiento de los algoritmos. Uno de los grandes problemas que nacen a partir de los algoritmos de búsqueda sistemática, es la continua aparición de valores individuales o conjuntos de ellos que no pueden participar de ninguna solución, situación conocida como inconsistencias locales (relacionado con el ejemplo anterior). Es por esto que nacen las técnicas llamadas de consistencia, o inferencia, que son utilizadas en los CSP con el objetivo directo de mejorar los métodos de búsqueda, reduciendo la cantidad de nodos a instanciar. Este tipo de técnicas elimina los valores inconsistentes de los dominios de las variables, logrando acotar el espacio de soluciones en un problema. La utilización de estos algoritmos no garantiza de ninguna forma que los escenarios posibles restantes sean derechamente soluciones al problema, pero si ha sido demostrado que la utilización de éstos como pre-proceso, reduce considerablemente la complejidad del problema.

### 4.3.1. Consistencia de Nodo

El nivel mas básico de consistencia es la de nodo o también llamada *nodo-consistencia*. Una variable $x_i$ es nodo-consistente si y sólo si todos los valores de su dominio $D_i$ son consistentes con las restricciones unarias sobre

la variable. Un *CSP* es nodo-consistente si y sólo si todas sus variables son nodo-consistentes. Si los valores contenidos en el dominio de la variable $x$ no satisfacen las restricciones, se está frente a una inconsistencia, donde el o los valores del dominio de la variable $x$ pueden ser eliminados, ya que no serían parte de ninguna solución.

$$\forall x_i \in X, \forall c_i \in C, \exists a \in D_i : (a) \in c_i$$

A continuación en el ejemplo de la Figura 8 se considera una variable $x$ con un dominio de $[2, ..., 10]$ y una restricción unaria $x \leq 5$. La técnica de *consistencia de nodo* se encargará de excluir todos aquellos valores del dominio de $x$ que no cumplan con la restricción.

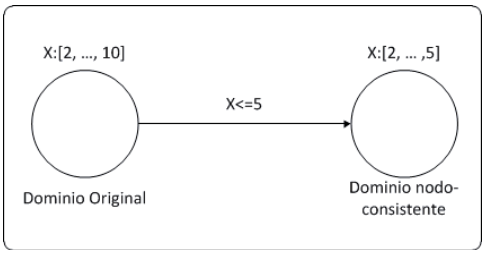



Figura 8: Consistencia de nodo.

### 4.3.2. Consistencia de Arco

A diferencia de la consistencia de nodo, la *consistencia de arco* trabaja con restricciones binarias. Un CSP es *arco-consistente* si para todo par de dominios $D_i$ y $D_j$, $y \neq z$, y una restriccion $R_n$ se cumple que:

$$\forall x_i \in D_y, \exists x_j \in D_z(x_i, xj) \; satisface \; R_n$$

Cumplir con lo antes dicho, es razón suficiente para eliminar cualquier valor del dominio $D_y$ de la variable $x_i$ que no sea arco-consistente, debido a que no sería parte de ninguna solución.

Como se puede observar en la Figura 9, para cada valor de $a \in$ [3,6], existe al menos un valor de $b \in$ [8,10] que satisface la restricción $x_i < x_j$, esto significa que el problema es arco-consistente, pero si la restricción hubiera sido $x_i = x_j$ el problema no hubiera sido arco-consistente.

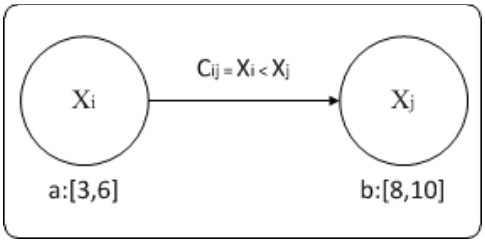



Figura 9: Consistencia de arco.

### 4.3.3. Consistencia de Camino

En las técnicas de inferencia local, el nivel superior a la Consistencia de Arco es la *Consistencia de Camino* o también llamada consistencia de senda (en inglés *Path-Consistency*), la cual fue propuesta por Ugo Montanari [17], la que consiste en que cada par de valores $a$ y $b$ de dos variables $x_i$ y $x_j$ se satisfaga la restricción entre ellas mediante la asignación de $a$ en $x_i$ y de $b$ en $x_j$ y además a lo largo del camino entre $x_i$ y$x_j$ exista para cada variable un valor de manera que se satisfagan todas las restricciones a lo largo del camino [6]. Cuando se logra satisfacer la consistencia de camino y además es arco-consistente y nodo-consistente, satisface fuertemente a la consistencia de camino (strongly path-consistent). De esta forma la consistencia de camino es satisfecha por un problema si y sólo si un par de variables $(x_i, x_j)$ es consistente de camino [5]:

$$\forall (a, b) \in C_{ij}, \forall x_k \in X, \exists c \in D_k$$

Tal que $c$ es un soporte para $a$ en $c_{ik}$ y para $b$ en $c_{jk}$.

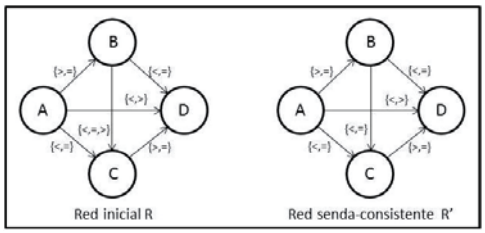



Figura 10: Ejemplo consistencia de camino.

En la Figura 10 se muestra un ejemplo de consistencia de camino, donde, a partir de una red inicial de restricciones R, se le aplica un algoritmo de

senda-consistencia, obteniendo una red senda-consistente R', donde cualquier camino entre cualquier par de nodos lo es.

### 4.3.4. K-Consistencia

Como se ha mencionado anteriormente nodo-consistencia (1 variable) se encarga de descartar aquellos valores de una variable que son inconsistentes ante una restricción, la arco-consistencia (2 variables) se encarga de eliminar aquellos valores inconsistentes de dos variables y la senda-consistencia (3 variables) excluye aquellos pares de valores consistentes que no se pueden extender de forma consistente a una tercera variable, en ese mismo orden la k-consistencia es una extensión de las consistencias locales, la cual envuelve a todas las consistencias (k-variables), a raíz de esto los niveles de consistencia dependen del número de variables que posean. En general, se dice que una red es k-consistente si y sólo si dada cualquier instanciación de k-1 variables, que satisfagan todas las restricciones entre ellas, existe al menos una instanciación de una variable k, tal que se satisfacen las restricciones entre las k variables [18].

### 4.3.5. Consistencia de Borde

En ocasiones chequear la consistencia de un problema se ve dificultado, ya que sus variables poseen dominios muy grandes o el conjunto de los valores que existen en tales dominios pertenecen a números reales o decimales. Debido a esto se ha propuesto la consistencia de borde, que es muy similar a la arco-consistencia, la que consiste en comprobar la arco-consistencia del problema a través de los límites (inferior y superior) de los dominios de las variables [7].

Se tiene un problema con dos variables $X$ e $Y$ con dominios $\{1; 8\}$ y $\{-2; 2\}$, respectivamente y la restricción $X = Y^3$. Este problema cumple la consistencia de borde, ya que los valores (1;8) para $X$ y (-2;2) para $Y$ son consistentes con la restricción. Sin embargo, el problema no es arco-consistente porque la instanciación $(Y; 0)$ es inconsistente, ya que no hay valores en D que satisfagan la restricción cuando $(Y; 0)$ [12].

## 4.4. Estrategias de Enumeración

Anteriormente se ha mencionado que un CSP está compuesto por dominios que están relacionados a un grupo o conjunto de variables, los que están sometidos bajo ciertas restricciones que tienen la finalidad de obtener alguna solución. Para obtener una de estas soluciones se utilizan tales restricciones,

las que están definidas sobre un conjunto de variables en el cual sus valores asignados son restringidos de tal manera que ninguna de estas restricciones sean quebrantadas. Para solucionar un CSP es necesario pasar por ciertas etapas, de forma alternada, llamadas de *propagación* y de *enumeración* y que en conjunto tienen como objetivo formar un árbol de búsqueda mediante un exhaustivo rastreo de los posibles valores que puedan ser asignados a las variables. La propagación consiste en podar el árbol de búsqueda de valores que no ayuden a encontrar alguna solución. La enumeración crea una rama instanciando una variable y crea otra rama cuando la primera no se ha podido satisfacer. Es en esta situación donde se deben solucionar dos circunstancias, una es de escoger la variable a enumerar a través de la utilización de una Heurística de selección de variables y la otra es de escoger el valor a asignar a dicha variable mediante una Heurística de selección de valor, ambas constituyen una *Estrategia de Enumeración*, muy importante al momento de mejorar el rendimiento para todo el proceso de resolución de un problema, es decir, menos tiempo en encontrar alguna solución [9].

### 4.4.1. Heurística de Selección de Variables

Esencialmente consiste en situar las variables de tal manera que la búsqueda sea más eficiente, es decir, ser capaz de detectar lo antes posible todos aquellos escenarios sin salida para minimizar el número de vueltas en el árbol de búsqueda para encontrar una solución. Algunas de estas heurísticas son:

- **First-Fail**: También llamada *Minimum Remaining Value*(MRV), se basa en un principio, el cual dice: *"Para tener éxito, trata primero donde es más propenso a fallar"* [10], esto quiere decir que es recomendable trabajar con la variable que tiene el menor número posible de opciones validas o el dominio más pequeño. Como ejemplo se considerará un caso del problema 8-reinas, donde las columnas son desde la letra $A$ a la $H$ y las filas desde el 1 al 8, se asignan reinas en las coordenadas *A1*, *B3* y *C5*, esto lleva a que la columna $D$ pueda tener 3 asignaciones posibles (*D2*, *D7* y *D8*), $E$ también pueda tener 3 asignaciones posibles, $F$ sólo una asignación posible y el resto de las columnas ($G$ y $F$) 3 asignaciones posibles cada una. Según ésta heurística donde se debería colocar una reina es en la columna $F$, ya que su dominio es más pequeño que el resto de las variables.

- **Most Constrained Variable**: Se basa en la elección de la variable que esté involucrada en el mayor número de restricciones de las variables

no asignadas. Con ésta heurística es posible solucionar problemas de 100-reinas.

- **Reduce-First**: Se basa en la elección de la variable con el dominio más grande. Esta heurística se caracteriza por ser la más adoptada en los casos que se poseen dominios continuos.

- **Round-Robin**: Basada en la elección de la variable en un orden equitativo y racional, es decir, se instancia una variable desde la primera hasta llegar a la última. Muy usado para el problema de organización de un torneo, donde todos los equipos deben enfrentarse entre sí.

### 4.4.2. Heurística de Selección de Valor

Consiste en elegir el valor del dominio de una variable que sea más conveniente para llegar a una o algunas soluciones, muchas de estas heurísticas tienen preferencia sobre el valor menos restringido de las variables, de esta manera los valores restantes serían aquellos que van a ser beneficiosos para las variables que lleven a alguna solución. Algunas de estas heurísticas son:

- **Min-Conflicts Value**: Esta heurística se basa en el siguiente argumento: "*Elige un valor que tenga el menor número de conflictos con valores asignados en otras variables. Si es que hay más de un valor para seleccionar, selecciona uno al azar*" [15]. Lo anteriormente mencionado se refiere básicamente a que se debe elegir el valor que produce la menor cantidad de conflictos para futuras asignaciones y si existe más de una debe ser elegida de forma aleatoria.

- **Smallest**: Se basa en la selección del valor más pequeño existente en el dominio de una variable. Útil para dominios donde prevalecen valores distantes entre ellos.

- **Median**: Heurística basada en la selección del valor que se encuentre posicionado dentro de los valores medios del dominio de alguna variable. Se utiliza de la misma manera en la que usa la heurística smallest.

- **Maximal**: Esta heurística, utilizada de la misma manera que las dos heurísticas mencionadas anteriormente, se basa en la selección del valor máximo del dominio de una variable.

# 5. Algoritmos de Arco Consistencia

Alcanzar la arco-consistencia trae como resultado la eliminación, sin mayor costo, de algunos valores que nunca formarán parte de alguna solución. Esta práctica es parte de las denominadas técnicas de inferencia incompleta, que a su vez, como fue mencionado en el punto 4.3 del presente informe, trabajan con restricciones asociadas a dos valores asignados a dos variables. Existen distintas definiciones entre valor y variable a la hora de referirse a la arco-consistencia, estos son [12]:

**Valor - Variable**: Un valor $a \in D_i$ es arco-consistente relativo a $X_j$ ssi existe un valor $b \in D_j$, tal que $\langle X_i, a\rangle$ y $\langle X_j, b\rangle$ satisfacen la restricción $R_{ij}$.

**Variable - Variable**: Una variable $X_i$ es arco-consistente relativa a $X_j$ ssi todos los valores en $D_i$ son arco-consistentes relativos a $X_j$. Por ejemplo una variable es arco-consistente si es consistente relativa a todas aquellas con las que interviene en alguna restricción.

**CSP completo**: Un *CSP* es arco-consistente ssi todas las variables son arco-consistentes, es decir, todos los arcos $R_{ij}$ y $R'_{ji}$ son arco-consistentes. Se debe tener en cuenta que en esta definición se habla de arco-consistencia completa. Un CSP es arco-inconsistente si una (o más) de sus variables no es(son) arco-consistente(s).

Como se ha mencionado, el principal objetivo de esta técnica es el hecho de restringir los dominios de dos variables asociadas a una restricción, el proceso secuencial mediante el cual se realiza esta operación se denomina *propagación* y para referirse a la forma en que los algoritmos realizan esta acción se utilizará el concepto de *granulidad*. El presente estudio se enfocará en cuatro algoritmos de arco-consistencia, AC-1, AC-2, AC-3 y AC-4, de los cuales los tres primeros son considerados de *grano-grueso* (en inglés coarse-grained algorithm), es decir, la propagación, al momento de eliminar un valor, se realizará solo a las variables relacionadas a las restricciones, en el caso del algoritmo AC-4 la propagación, al momento de eliminar un valor, se realizará solo a los valores relacionados a las variables correspondientes, es decir, *grano fino* (en inglés fine-grained algorithm). En términos cronológicos los algoritmos se presentan tal como se muestran en la Figura 11, donde las flechas indican la procedencia de cada algoritmo.

A continuación serán especificados los algoritmos antes mencionados, los

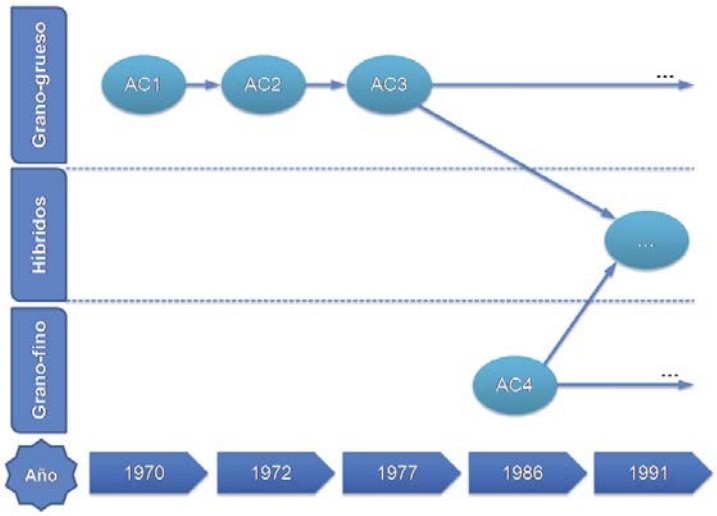



Figura 11: Cronología y granulidad de los distintos algoritmos AC.

que son de vital importancia para la investigación del presente informe.

## 5.1. Algoritmo AC-1

Algoritmo propuesto por Mackworth en los años 70', posee una complejidad espacial $O\left(n^3d^3\right)$ y una complejidad temporal $O\left(ned^3\right)$, donde $n$ son las variables, $e$ los arcos y $d$ la talla máxima del dominio. AC-1 está compuesto por su procedimiento principal y dos sub-procesos, NC y REVISAR, los que también son utilizados por algunos de los algoritmos posteriores de arco-consistencia. A continuación se presenta el algoritmo AC-1 y sus sub-procesos.

**Algoritmo 1** Procedimiento Nodo-Consistencia

1: $PROCEDIMIENTO\ NC\left(i\right)$
2: $D_i \leftarrow D_i \cap \{x|P_i\left(x\right)\}$
3: inicio
4: **para** $i \leftarrow 1$ hasta $n$ **hacer**
5: $\quad NC\left(i\right)$
6: **fin para**
7: fin inicio

Además se define la función *REVISAR*, la cual se especifica de la siguiente manera [13]:

Dados dos dominios discretos, $D_i$ y $D_j$, para dos variables $v_i$ y $v_j$, tal que son nodos consistentes. Si $x \in D_i$ y no hay un valor $y \in D_j$, tal que cumpla con la restricción $R_{ij}\,(x, y)$, entonces $x$ puede ser borrado del dominio $D_i$. Cuando lo anterior ha sido logrado para cada valor existente en $D_i$, es posible decir que el arco$(i, j)$ es consistente, pero no necesariamente el arco inverso $(j, i)$. Esta parte del proceso se detalla en el siguiente algoritmo:

**Algoritmo 2** Procedimiento REVISAR

**Entrada:** CSP $P'$ definido por dos variables $X = \{X_i, X_j\}$, dominios $D = \{D_i, D_j\}$ y una restricción $R = \{R_{ij}\}$.

**Salida:** change = **verdadero** si el dominio fue modificado y change = **falso** cuando no ha sido modificado.

```
1: inicio
2: change ← falso
3: para cada a ∈ D_i hacer
4:    si ¬∃b ∈ D_j tal que (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij entonces
5:       Eliminar a de D_i
6:       change ← verdadero
7:    fin si
8: fin para
9: devolver change
10: fin inicio
```

A continuación se presenta el pseucódigo de AC-1, el cual retorna un CSP arco-consistente, donde se emplea la función *REVISAR* las veces que sea necesario para lograr que todos los arcos pertenecientes al grafo de restricciones $G$ sean consistentes, ya que cumple la función de eliminar todos aquellos valores del dominio $D_j$ que no sean partícipes de ninguna solución, es decir, valores *redundantes* o inconsistentes. La función REVISAR retorna un *verdadero* en el caso de que se haya eliminado algún valor redundante, en el caso contrario retorna un *falso*, ya que el arco es consistente [14].

**Algoritmo 3** Pseudocódigo AC-1

```
1: inicio
2: para i ← 1 hasta n hacer
3:    NC(i)
4: fin para
5: Q ← {(i,j) | (i,j) ∈ arcos(G), i ≠ j}
6: repetir
7:    CHANGE ← false
8:    para cada (i,j) ∈ Q hacer
9:       CHANGE ← (REVISAR((i,j)) ∨ CHANGE)
10:   fin para
11: hasta que ¬CHANGE
12: fin inicio
```

Este algoritmo es considerado ineficiente, ya que recorre todas las restricciones en cada ocasión que se encuentre una inconsistencia o cuando un dominio ha sido alterado.

## 5.2. Algoritmo AC-2

Algoritmo propuesto por Mackworth al igual que AC-1, basándose en otro proceso de filtrado implementado por Waltz [20], donde básicamente se logra que la red sea consistente con sólo una pasada a través de los nodos del grafo de restricciones, donde Q y Q' son dos colas encargadas de mantener los arcos directos e inversos. Posee una complejidad espacial $O(e)$ y una complejidad temporal $O(ed^3)$, donde $e$ son los arcos y $d$ la talla máxima del dominio.

**Algoritmo 4** Pseudocódigo AC-2

```
1: inicio
2: para i ← 1 hasta n hacer
3:    NC (i)
4:    Q ← {(i, j) | (i, j) ∈ arcos (G) , j < i}
5:    Q' ← {(j, i) | (j, i) ∈ arcos (G) , j < i}
6:    mientras Q no este vacío hacer
7:       mientras Q no este vacío hacer
8:          pop (k, m) desde Q
9:          si REVISAR((k, m)) entonces
10:            Q' ← Q' ∪ {(p, k) | (p, k) ∈ arcos (G) , p ≤ i, p ≠ m}
11:         fin si
12:         Q ← Q'
13:         Q ← ∅
14:      fin mientras
15:   fin mientras
16: fin para
17: fin inicio
```

AC-2 es mas eficiente que el algoritmo AC-1, ya que sólo recorre aquellos nodos que estén ligados a la inconsistencia. Aún así es ineficiente, ya que al aplicar REVISAR recorrerá algunos arcos que ya han sido extraídos de las colas Q y Q'.

## 5.3. Algoritmo AC-3

Algoritmo propuesto por Mackworth, al igual que AC-1, el cual se destaca por la simpleza en sus estructuras, ya que utiliza solamente una *cola*, la que va almacenando las restricciones que serán evaluadas. Como se ha mencionado anteriormente AC-3 pertenece a la categoría de algoritmos de *grano grueso*, debido a que realiza sus propagaciones a nivel de restricciones, además posee una complejidad espacial $O(e)$ y una complejidad temporal $O(ed^3)$, donde $e$ son los arcos y $d$ la talla máxima del dominio. Gracias a su sencillez en el funcionamiento es muy utilizado, incluso ha sido base fundamental de una serie de algoritmos que se han propuesto en entre los años 1996 al 2007 [11].

**Algoritmo 5** Pseudocódigo AC-3

**Entrada:** CSP, $P = (X, D, R)$

**Salida:** **verdadero** y $P'$ el cual es arco-consistente o **falso** y $P'$ el cual es arco-inconsistente ya que su dominio quedo vacío

```
1: inicio
2: para cada arco R_ij ∈ R hacer
3:    Añadir(Q, R_ij)
4:    Añadir(Q, R_ji)
5: fin para
6: mientras Q ≠ ∅ hacer
7:    Seleccione y borre R_ij de la cola de Q
8:    si REVISAR(R_ij)=verdadero entonces
9:       si D_i ≠ ∅ entonces
10:         Añadir(Q, R_ki) con k ≠ i, k ≠ j
11:      si no
12:         devolver falso (dominio vacio)
13:      fin si
14:   fin si
15: fin mientras
16: devolver verdadero
17: fin inicio
```

En relación al funcionamiento del algoritmo AC-3 se comienza con añadir parte de las restricciones a la cola. Luego comienza un ciclo donde se extrae el primer elemento de la cola y sobre ese elemento se aplica el algoritmo REVISAR, el que dependiendo de que si se altera o no el dominio, retorna un valor booleano (falso o verdadero). Si el valor retornado es *verdadero* y además el dominio de una variable no es vacío, se procede a añadir una nueva restricción a la cola, luego se sigue realizando el ciclo hasta que todos los elementos de la cola sean extraídos, es decir, cuando la cola quede vacía. Si el dominio de alguna variable se encuentra vacío, quiere decir que el problema no tiene solución. A pesar de haber mejorado ciertas características de los algoritmos anteriores, éste cae en ciertas redundancias en la hora de utilizar el algoritmo REVISAR ya que no posee ningún método de almacenamiento mediante el cual se verifique que las restricciones que entran no lo hayan hecho anteriormente.

## 5.4. Algoritmo AC-4

Propuesto por Roger Mohr y Thomas C. Henderson [16] en los años 80'. Caracterizado por realizar un proceso distinto al de sus antecesores, ya que antes de realizar la arco-consistencia, se encarga de analizar todas las restricciones y sus respectivas asignaciones o instanciaciones, guardando todos aquellos valores que causen inconsistencias, convirtiéndolo en uno de los procesos que guarda mayor información al momento del pre-proceso de la arco-consistencia, información necesaria para evitar el chequeo recursivo de valores ya revisados durante la poda, por esto se ha catalogado al algoritmo AC-4 como un algoritmo de grano fino o fine-grained algorithm, a diferencia de los anteriores que son de grano grueso, porque las propagaciones las realiza a nivel de valores y no a nivel de restricciones. Este procedimiento utiliza estructuras de datos más complejas que las de sus predecesores, pero a la vez muy útiles, ya que éstas ayudan a que AC-4 posea una complejidad óptima de $O\left(ed^2\right)$ y una complejidad espacial de $O\left(ed^2\right)$, donde $e$ es el número de restricciones y $d$ es la talla máxima del dominio, son estas características las que hacen que AC-4 sea uno de los algoritmos más utilizados (junto con AC-3) en el estudio de la de la arco-consistencia. Las estructuras de datos a utilizar se detallan a continuación:

- $S$ es una matriz $S\left[X_j, b\right]$ que contiene la lista de instanciaciones $\langle X_i, a\rangle$ soportadas por $\langle X_j, b\rangle$.
- *Counter* es una matriz $Counter\left[X_i, a, X_j\right]$ que contiene el número de soportes para el valor $a \in D_i$ en la variable $X_j$.
- $M$ es una matriz $M\left[X_i, a\right]$ que contiene el valor 1 si el valor $a \in D_i$, o contiene el valor 0 si el valor $a \notin D_i$.
- $Q$ es una cola que almacena tuplas $\langle X_i, a\rangle$ (instanciaciones eliminadas) que requiere un procesamiento posterior.

**Algoritmo 6** Pseudocódigo AC-4

**Entrada:** CSP, $P = (X, D, R)$

**Salida:** **verdadero** y $P'$ el cual es arco-consistente o **falso** y $P'$ el cual es arco-inconsistente

```
1: inicio
2: InicializarAC4(P)
3: si initial = verdadero entonces
4:    mientras Q ≠ ⊘ hacer
5:       Seleccione y elimine ⟨X_j, b⟩ de la cola Q
6:       para cada ⟨X_i, a⟩ ∈ S[X_j, b] hacer
7:          Counter[X_i, a, X_j] ← Counter[X_i, a, X_j] − 1
8:          si Counter[X_i, a, X_j] = 0 ∧ M[X_i, a] = 1 entonces
9:             Elimine a de D_i
10:            Q ← Q ∪ ⟨X_i, a⟩
11:            M[X_i, a] ← 0
12:         fin si
13:         si D_i ≠ ⊘ entonces
14:            devolver falso
15:         fin si
16:      fin para
17:   fin mientras
18:   devolver verdadero
19: si no
20:   devolver falso
21: fin si
22: fin inicio
```

**Algoritmo 7** Pseudocódigo InicializarAC4

**Entrada:** CSP, $P = (X, D, R)$ donde $R$ contiene a todas las restricciones tanto directas como inversas

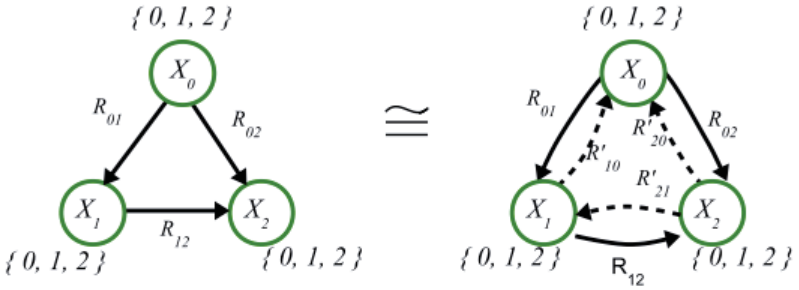
**Salida:** Una variable booleana *initial*(donde *initial*=**verdadero** cuando ningún dominio es vacio y *initial*=**falso** en el caso contrario); cola$Q$; matrices $S, M, Counter$ y CSP $P'$ actualizados

1: inicio
2: $Q \leftarrow \{\}$
3: $S\left[X_j, a\right] \leftarrow \{\}\ \{\forall X_j \in X \wedge \forall b \in D_j\}$
4: $M\left[X_i, a\right] \leftarrow \{\}\ \{\forall X_i \in X \wedge \forall b \in D_i\}$
5: $Counter\left[X_i, a, X_j\right] \leftarrow 0\ \{\forall X_{ij}, X_j \in X, i \neq j \wedge \forall a \in D_i\}$
6: **para** cada arco$R_{ij} \in R$ **hacer**
7:  **para** cada valor $a \in D_i$ **hacer**
8:  $total \leftarrow 0$
9:  **para** cada valor $b \in D_j$ **hacer**
10:  **si** $(\langle X_i, a\rangle, \langle X_j, b\rangle) \in R_{ij}$ **entonces**
11:  $total \leftarrow total + 1$
12:  Añadir $(S\left[X_j, b\right], \left[X_i, a\right])$
13:  **fin si**
14:  **fin para**
15:  **si** $total = 0$ **entonces**
16:  Eliminar $a$ de $D_i$
17:  $Q \leftarrow Q \cup \langle X_i, a\rangle$
18:  $M\left[X_i, a\right] \leftarrow 0$
19:  **si no**
20:  $Counter\left[X_i, a, X_j\right] \leftarrow total$
21:  **fin si**
22:  **si** $D_i = \oslash$ **entonces**
23:  **devolver** $initial \leftarrow falso$
24:  **fin si**
25:  **fin para**
26: **fin para**
27: **devolver** $initial \leftarrow verdadero$
28: fin inicio

El procedimiento principal AC-4 se divide en 2 fases: una de inicialización y otra de propagación. La fase de inicialización consiste en que las estructuras $Q$ y $S$ tomen un valor vacío y las matrices $M$ y $Counter$ se les asigne valores 1 y 0 respectivamente, para luego recorrer los dominios $D_i$ y $D_j$ de cada arco

$R_{ij}$ en búsqueda de aquellos valores $b$ que sean soporte de $a$, si esto se cumple se almacenan en la matriz $S\,[X_j, b]$. En el caso de que no existan soportes, el valor $a$ es eliminado de $D_i$, es almacenado en la cola $Q$ y la matriz $M$ contendrá el valor 0, de lo contrario en la matriz *Counter* se almacenará la cantidad de soportes que posee el valor $a$ en la variable $X_j$ para finalmente devolver *initial=verdadero*. Como caso excepcional si el dominio $D_i$ es vacío, se retornará *initial* = *falso*, lo que indica que el problema no tiene solución. En la fase de propagación el papel fundamental lo ocupa la cola $Q$, ya que si ésta no es vacía, se presentarán dos situaciones, en la primera el algoritmo asegura que todos los valores de los dominios son consistentes con todas las restricciones (son arco-consistentes), mientras que la segunda, el algoritmo indica que el problema no tiene solución.

En las siguientes tablas (ver Tabla 1, Tabla 2, Tabla 3, Tabla 4) se muestra el procedimiento, utilizando AC-4, para el ejemplo de la Figura 12.




**Valores iniciales de $R$**

| Restricción | | |
|---|---|---|
| $R_{ij}$ | $X_i$ op $X_j$ | dirección |
| $R_{01}$ | $X_0 < X_1$ | directa |
| $R'_{10}$ | $X_1 > X_0$ | inversa |
| $R_{02}$ | $X_0 < X_2$ | directa |
| $R'_{20}$ | $X_2 > X_0$ | inversa |
| $R_{12}$ | $X_1 < X_2$ | directa |
| $R'_{21}$ | $X_2 > X_1$ | inversa |

Figura 12: Ejemplo de CSP binario normalizado con tres variables.

| Iter | Restricción $R_{ij} : X_i op X_j$ | valor a | valor b | variable total | Poda $X_i$ | Add Q |
|---|---|---|---|---|---|---|
| 1 | $R_{02} : X_0 < X_2$ | 0 | 0 | 0 | | |
| | | | 1 | 1 | | |
| | | | 2 | 2 | | |
| | | 1 | 0 | 0 | | |
| | | | 1 | 0 | | |
| | | | 2 | 1 | | |
| | | 2 | 0 | 0 | | |
| | | | 1 | 0 | | |
| | | | 2 | 0 | $X_0 = 2$ | $\langle X_0, 2\rangle$ |
| 2 | $R'_{20} : X_2 > X_0$ | 0 | 0 | 0 | | |
| | | | 1 | 0 | $X_2 = 0$ | $\langle X_2, 0\rangle$ |
| | | 1 | 0 | 1 | | |
| | | | 1 | 1 | | |
| | | 2 | 0 | 1 | | |
| | | | 1 | 2 | | |
| 3 | $R_{01} : X_0 < X_1$ | 0 | 0 | 0 | | |
| | | | 1 | 1 | | |
| | | | 2 | 2 | | |
| | | 1 | 0 | 0 | | |
| | | | 1 | 0 | | |
| | | | 2 | 1 | | |
| 4 | $R'_{10} : X_1 > X_0$ | 0 | 0 | 0 | | |
| | | | 1 | 0 | $X_1 = 0$ | $\langle X_1, 0\rangle$ |
| | | 1 | 0 | 1 | | |
| | | | 1 | 1 | | |
| | | 2 | 0 | 1 | | |
| | | | 1 | 2 | | |
| 5 | $R_{12} : X_1 < X_2$ | 1 | 1 | 0 | | |
| | | | 2 | 1 | | |
| | | 2 | 1 | 0 | | |
| | | | 2 | 0 | $X_1 = 2$ | $\langle X_1, 2\rangle$ |
| 6 | $R'_{21} : X_2 > X_1$ | 1 | 1 | 0 | $X_2 = 1$ | $\langle X_2, 1\rangle$ |
| | | 2 | 1 | 1 | | |

Tabla 1: Iteraciones realizadas por el procedimiento InicializarAC4 para el ejemplo mostrado en la Figura 12.

| $M\,[var, val]$ | Inicio | Inicializar | Propagación |
|---|---|---|---|
| $M\,[X_0, 0]$ | 1 | | |
| $M\,[X_0, 1]$ | 1 | | 0 |
| $M\,[X_0, 2]$ | 1 | 0 | |
| $M\,[X_1, 0]$ | 1 | 0 | |
| $M\,[X_1, 1]$ | 1 | | |
| $M\,[X_1, 2]$ | 1 | 0 | |
| $M\,[X_2, 0]$ | 1 | 0 | |
| $M\,[X_2, 1]$ | 1 | 0 | |
| $M\,[X_2, 2]$ | 1 | | |

Tabla 2: Cambios en la matriz M después de las etapas de Inicialización y de Propagación realizadas por el procedimiento AC4 para el ejemplo de la Figura 12.

| $Counter\,[X_i, a, X_j]$ | Inicio | Inicializar | Propagación |
|---|---|---|---|
| $Counter\,[X_0, 0, X_2]$ | 0 | 2 | 1 |
| $Counter\,[X_0, 1, X_2]$ | 0 | 1 | |
| $Counter\,[X_0, 2, X_2]$ | 0 | 0 | |
| $Counter\,[X_0, 0, X_1]$ | 0 | 2 | 1 |
| $Counter\,[X_0, 1, X_1]$ | 0 | 1 | 0 |
| $Counter\,[X_0, 2, X_1]$ | 0 | | |
| $Counter\,[X_1, 0, X_0]$ | 0 | 0 | |
| $Counter\,[X_1, 1, X_0]$ | 0 | 1 | |
| $Counter\,[X_1, 2, X_0]$ | 0 | 2 | 1 |
| $Counter\,[X_1, 0, X_2]$ | 0 | | |
| $Counter\,[X_1, 1, X_2]$ | 0 | 1 | |
| $Counter\,[X_1, 2, X_2]$ | 0 | 0 | |
| $Counter\,[X_2, 0, X_0]$ | 0 | 0 | |
| $Counter\,[X_2, 1, X_0]$ | 0 | 1 | |
| $Counter\,[X_2, 2, X_0]$ | 0 | 2 | 1 |
| $Counter\,[X_2, 0, X_1]$ | 0 | | |
| $Counter\,[X_2, 1, X_1]$ | 0 | 0 | |
| $Counter\,[X_2, 2, X_1]$ | 0 | 1 | |

Tabla 3: Cambios en la matriz Counter después de las fases de Inicialización y de Propagación realizadas por el procedimiento AC4 para el ejemplo de la Figura 12.

| $S[X_j, b]$ | Inicio | Inicializar |
|---|---|---|
| $S[X_0, 0]$ | | $\{\langle X_2, 1\rangle, \langle X_2, 2\rangle, \langle X_1, 1\rangle, \langle X_1, 2\rangle\}$ |
| $S[X_0, 1]$ | | $\{\langle X_2, 2\rangle, \langle X_1, 2\rangle\}$ |
| $S[X_0, 2]$ | | |
| $S[X_1, 0]$ | | |
| $S[X_1, 1]$ | | $\{\langle X_0, 0\rangle, \langle X_2, 2\rangle\}$ |
| $S[X_1, 2]$ | | $\{\langle X_0, 0\rangle, \langle X_0, 1\rangle\}$ |
| $S[X_2, 0]$ | | |
| $S[X_2, 1]$ | | $\{\langle X_0, 0\rangle\}$ |
| $S[X_2, 2]$ | | $\{\langle X_0, 0\rangle, \langle X_0, 1\rangle, \langle X_1, 1\rangle\}$ |

Tabla 4: Cambios en la matriz S después de las fases de Inicialización y de Propagación realizadas por el procedimiento AC4 para el ejemplo de la Figura 12.

# 6. RSSolver y ANTLR

Hasta ahora se han mencionado conceptos sobre CSPs y de lo que engloba, tales como algoritmos de búsqueda y técnicas de consistencia. Además es importante mencionar que se ha utilizado un solver hecho por el Dr. Ricardo Soto, donde para poder trabajar con un modelo, es necesario llevarlo a un código, del cual se pueda trabajar libremente con los elementos de este, como sus restricciones, variables y dominios. El lenguaje utilizado por el solver es Java, permitiendo trabajar con los elementos del modelo como clases y realizar distintos métodos, necesarios para realizar los algoritmos, tanto de búsqueda como de consistencia. Ahora todo el proceso, desde el modelo a trabajarlo a lenguaje Java, se puede realizar gracias a la utilización de otra herramientas, que permite crear lenguajes y trabajar en ellos en un lenguaje deseado, ANTLR.

El desglose de sus siglas permite acercarse un poco a su significado, **AN**other **T**ool for **L**anguage **R**ecognition, que en español seria *Otra Herramienta para el Reconocimiento de Lenguaje*. Es una herramienta que opera sobre lenguajes, y que a partir de la descripción gramatical de estos, es posible construir parsers (reconocedores), intérpretes, compiladores y traductores de lenguajes. En el proceso de definición de lenguajes es posible encontrar una serie de pasos, los cuales ayudan a llegar al fin último de la herramienta, la creación o definición de un lenguaje y su posterior reconocimiento.

## 6.1. Analizador Léxico

La definición formal indica que el léxico es un conjunto de palabras (diccionario) de un idioma, esta misma definición se aplica en este caso, ya que como bien se sabe, cada lenguaje de programación posee restricciones en cuanto a los elementos utilizables a la hora de programar, ya sean símbolos, letras, números u otros.

```
protected NUEVA_LINEA : “ \r \n ”
{ newline() ; };
BLANCO : (‘  ‘|’ \t ’|NUEVA_LINEA)
{ $setType(Token.SKIP) ; };
protected DIGITO : ‘ 0 ’ .. ‘ 9 ’;
NUMERO : (DIGITO)+ (‘ . ’(DIGITO) + ) ? ;
OPERADOR : ‘  + ‘|’ – ‘|’ / ‘|’ * ‘ ;
PARENTESIS : ‘ ( ’|’ ) ’ ;
SEPARADOR : ‘ ; ’ ;
```

Figura 13: Definición del léxico en ANTLR.

En el ejemplo mostrado en el código anterior se puede observar la definición de 7 elementos de un lenguaje, si bien la comprensión a simple vista es bastante sencilla, se deben destacar algunos elementos que son utilizados para acciones específicas. Existen símbolos dentro de la notación que permiten darle un sentido *dinámico* a cada una de las reglas:

- *, cero o muchos.
- +, uno o muchos.
- ?, opcional.
- |, opción.

Estos elementos tomarán dicho significado cuando no estén entre comillas. Por otra parte como se puede ver en el ejemplo, los rangos de caracteres son especificados a través de una simbología especifica (". . ") . Además es posible destacar ciertas acciones para una regla, como es el caso de `$setType(Token.SKIP);`, que permite que el *símbolo* definido o *token* sea obviado por el analizador léxico.

## 6.2. Analizador Sintáctico

En este caso, la sintaxis corresponde a una serie de reglas que definen las secuencias correctas de los elementos de un lenguaje de programación. Si se toma como parte de un proceso el analizador sintáctico o *parser*, sería la segunda etapa en donde se reciben los tokens definidos en el léxico, se revisa la sintaxis en relación a las reglas definidas y se genera un árbol denominado *AST*, que es una estructura de análisis.

```
program: VAR_RW! var_dec BEGIN_RW body END_RW!
         {## = #( #[PROGRAM, "PROGRAM"] ,##);};

var_dec: (type IDENT SEMICOLON!)*
         {## = #( #[VAR_DEC, "VAR_DEC"] ,##);};

type:    NUMERIC_TYPE|STRING_TYPE;

program: IDENT ASSIG expr SEMICOLON!
         {## = #( #[ASSIGN, "ASSIGN"] ,##);};
```

Figura 14: Analizador sintáctico en ANTLR.

Lo primero a destacar en esta parte del proceso es que cada regla está definida con minúsculas y cada token o palabra reservada utilizada para cada regla está con mayúsculas. Cada regla puede formar parte de la definición de otra, por ejemplo en la definición de *program* se utiliza *var_ dec*, la cual es una regla definida posteriormente. Como se aprecia en la parte inferior de algunas reglas, se puede ver la línea `{## = # ( # [PROGRAM, "PROGRAM"] , ##) ; };` , ayuda a formar y ordenar el árbol AST mencionado anteriormente y que se muestra en el la Figura 15.

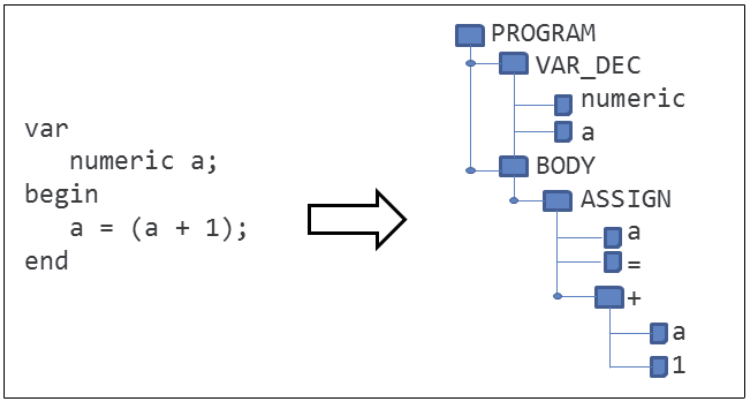



Figura 15: Ejemplo de generación de árbol AST en ANTLR.

## 6.3. Analizador Semántico

En el análisis léxico se detectaban los tokens. En el análisis sintáctico se agrupaban dichos tokens en reglas sintácticas, y se construía un primer AST.

El cometido del análisis semántico es doble:

- Por un lado debe detectar errores semánticos (incoherencias con respecto a las reglas semánticas del lenguaje). Este tipo de errores son muy recurrentes a la hora de programar y son variados los tipos de errores que se pueden encontrar. Cuando una o varias expresiones son incompatibles a nivel de operador, identificadores mal escritos o problemas de contexto de las variables, son algunos ejemplos de este tipo de errores.
- Por otro, debe enriquecer el AST con información semántica que será necesaria para la generación de código.

Los errores semánticos a los que nos estamos refiriendo aquí son los errores semánticos estáticos, que son aquellos detectados durante la compilación (en el análisis semántico). En oposición a los errores semánticos estáticos están los errores semánticos dinámicos, que se detectan durante la ejecución del programa ya compilado (y no forman parte del análisis semántico propiamente dicho) [8].

```
program: #(PROGRAM var_dec body);

var_dec: #(VAR_DEC (type id:IDENT {sI.addVar(id);}) *);

type:    NUMERIC_TYPE|STRING_TYPE;

body:    #(BODY assign);

assign:  #(ASSIGN ( id:IDENT {sI.checkVar(id); } ASSIG expr));
```

Figura 16: Analizador semántico en ANTLR.

En el código expuesto sobre el presente párrafo (Figura 16) es posible ver cómo desde el código ANTLR utilizado en la etapa anterior, se envían elementos al analizador semántico, que en este caso corresponde a los métodos encontrados en **sI**, como **addVar** y **checkVar**, a los que se les envían las variables de tipo AST para realizar el análisis. **addVar** añade las variables declaradas a una tabla, siempre y cuando la variable no haya sido registrada con anterioridad, ya que no pueden haber dos variables con el mismo nombre. Complementando la función anterior se tiene **checkVar**, que revisa que la variable que se utiliza en un determinado momento haya sido definida con

anterioridad, de esta forma se completa el proceso de chequeo de variables comprobando cada uno de los casos posibles

# 7. Implementación

Como ya ha sido expresado, el objetivo principal del presente proyecto es implementar técnicas de filtraje en algoritmos de búsqueda incluidos en un prototipo de Solver. La teoría ha mostrado que existen distintos algoritmos o técnicas para mejorar el proceso de búsqueda en los distintos problemas asociados a la programación con restricciones, pero de la teoría a la implementación existe un gran trecho, más aún cuando parte de lo implementado ha sido desarrollado por el Dr. Ricardo Soto, previa a la actual investigación, es por ello que en esta etapa del proyecto se ha realizado un detallado estudio del Solver denominado *RSSolver*.

## 7.1. Mejoras de AC-1

La investigación realizada hasta ahora ha sido para conocer y nutrir el conocimiento, con respecto al mundo de la programación con restricciones, lo que va a ser, desde ahora, muy útil para entender el funcionamiento del algoritmo de consistencia realizado en éste proyecto.

Como ya se vio anteriormente, AC-1 es el primer algoritmo de consistencia que surgió, en lo que respecta a los estudios de la programación con restricciones, es por lo cual se consideró para el presente proyecto, ya que actualmente el solver RSSolver, no posee ningún tipo de algoritmos de arco-consistencia, teniendo por objetivo, realizar un algoritmo de arco consistencia para éste solver.

En primera instancia se realizó un algoritmo AC-1, basándose en las bases de la teoría del algoritmo AC-1 [13], el cual se caracteriza por realizar una búsqueda de inconsistencia única, es decir, que se detiene al encontrar una inconsistencia y si desea buscar más inconsistencias, podrá volver a revisar, desde el principio, las inconsistencia. Esto es ya que AC-1 es caracterizado por su ineficiencia, ya que no busca todas las inconsistencias, sino que solo lo hace para una única inconsistencia que pueda haber en algún modelo.

En el presente proyecto se realizó, además de AC-1, un algoritmo de consistencia que mejora a este último, caracterizándolo por su velocidad en encontrar la solución y su eficacia en encontrar inconsistencias.

El nuevo algoritmo posee las siguientes potencialidades, que la diferencian de AC-1, ya que mejoran ciertas características de éste y a la vez potencian aún más el algoritmo, las cuales son:

- Búsqueda de más de una inconsistencia (dependerá del modelo utilizado).

- Búsqueda bidireccional.
- Búsqueda recursiva.

Con respecto a la búsqueda de más de una inconsistencia, se refiere a que si el algoritmo logra toparse con una inconsistencia, este no va a detenerse ni volver a comenzar desde el principio del modelo, la existencia de inconsistencias, sino que buscará hasta recorrer todo el árbol de búsqueda, encontrando todas las inconsistencias que se le presenten.

Con respecto a la búsqueda bidireccional, se refiere a que en el modelo se tienen restricciones, las cuales son binarias, por ejemplo $x + y < 5$, donde en este caso la variable $x$ es la base y la variable $y$ el soporte, esto quiere decir, que las instanciaciones se harán de manera que se va a instanciar primero una variable $x$ con un valor del dominio y esta se va a evaluar en la restricción con todas las instanciaciones de soporte, como se observa en la Figura 17. Es por ende, que al algoritmo, se le dio la capacidad de realizar están instanciaciones, ya sea con la variable $x$ como base o la variable $y$ como base, con sus respectivos soportes (base $x$ con soporte $y$, base $y$ con soporte $x$), es decir, bidireccionalidad.

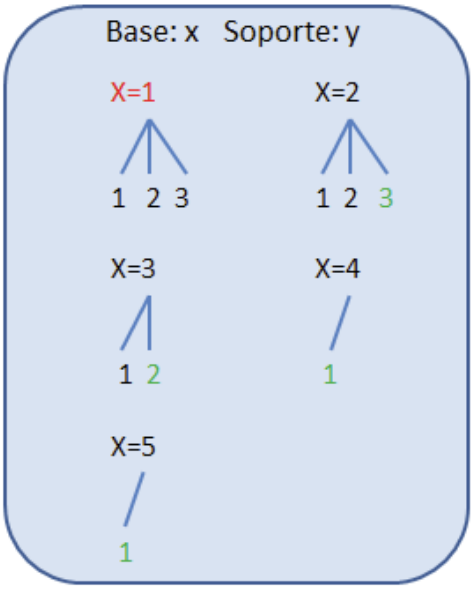



Figura 17: Ejemplo instanciaciones variable base con sus soportes, en algoritmo mejorado de AC-1.

Búsqueda recursiva se refiere a que el algoritmo recorrerá el árbol de búsqueda, las veces que sea necesaria, hasta que ya no existan instanciaciones, de esta manera logrando un CSP mínimo equivalente, es decir, un CSP el cual va a ser equivalente a su original, pero sin inconsistencias. Para lograr esto el algoritmo si encuentra inconsistencias, automáticamente revisara de nuevo el árbol de búsqueda, ya que si se eliminan valores inconsistentes de los dominios, podrían generar nuevas inconsistencias, es a raíz de esto, que el algoritmo revisara el árbol de búsqueda hasta el punto que ya no existan inconsistencias, de esta manera asegurando que no existan inconsistencias en el modelo.

Cabe destacar que todas estas características fueron realizadas en RSSolver, el cual en el capitulo a continuación se detallará mas a fondo la implementación del algoritmo mejorado.

## 7.2. RSSolver

Un *Solver* es una herramienta utilizada para resolver CSPs, es decir, que involucran variables, dominios y restricciones. En este caso específico la investigación se ha realizado sobre el prototipo de Solver *RSSolver*, desarrollado por el profesor Dr. Ricardo Soto De Giorgis. Este Solver incorpora un lenguaje desarrollado específicamente para crear los modelos correspondientes a un problema específico, en donde se deben incorporar los 3 elementos claves mencionados anteriormente. A continuación se muestra un ejemplo de modelo escrito en el lenguaje creado para RSSolver:

$$
\begin{array}{c}
variables: \\
\hline
x\ in\ [1, 1000]; \\
y\ in\ [1, 8]; \\
z\ in\ [1, 8]; \\
\\
constraints: \\
\hline
x >= 3; \\
x + y + z > 800; \\
z > 3;
\end{array}
$$

Como se puede ver en el ejemplo anterior, en el lenguaje se deben especificar claramente las variables (x, y, z) con sus respectivos dominios, que deben ser detallados luego de la palabra reservada *in* entre paréntesis cuadrado.

Luego de esto, se deben escribir las restricciones que deben estar asociadas a las variables ya declaradas.

La idea o propósito de definir un lenguaje de modelado es ayudar a que el usuario se aleje lo más posible de la engorrosa etapa de codificación, pudiendo de esta manera expresar un problema de forma comprensible utilizando una semántica y una sintaxis mucho más amigable y simplificada. Cabe mencionar que en este lenguaje de modelado, los dominios deben ser definidos a partir de sus cotas (inferior y superior), esto puede ser un impedimento a la hora de querer definir un dominio del tipo $[1, 26, 115]$, es decir con números no consecutivos, pero una gran ventaja para definir dominios amplios, como el asignado a la variable $x$ en el ejemplo anterior.

### 7.2.1. Modelo y Compilación

En los párrafos anteriores se ha explicado a grandes rasgos la herramienta sobre la cual se desarrolla el proyecto, pero existe una estructura detrás del funcionamiento del Solver. A través del lenguaje de modelado presentado anteriormente se crea un modelo asociado a un problema que se quiere resolver. Este modelo debe pasar primero por un proceso de compilación (rsSolver.compiler.*). El primer paso en la etapa de compilación es el análisis léxico (rsSolverLexer.g), a través del cual se revisa cada uno de los símbolos y palabras reservadas utilizadas en el modelo. A continuación se expone parte del código del analizador léxico del Solver(Figura 18).

```
OP_ASIG
options { paraphrase="assignment operator (':=')"; }     : ":="  ;

OP_PLUS
options { paraphrase="plus operator ('+')"; }            : "+"  ;

OP_PLUS_PLUS
options { paraphrase="increment operator ('++')"; }      : "++"  ;

OP_MINUS
options { paraphrase="minus operator ('-')"; }           : "-"   ;
```

Figura 18: Porción del analizador léxico en RSSolver.

El código anterior muestra cuatro de los símbolos utilizables en el RSSolver (asignación, suma, incremento y resta). A diferencia de lo que se explica en el capítulo 5 sobre el análisis léxico, se puede visualizar la palabra *options*, la que tiene una función explicativa para los mensajes de error mostrados

en la consola. Tras un paso exitoso por el análisis léxico, comienza el análisis sintáctico (rsSolverParser.g), en esta etapa se revisa la estructura del modelo a partir de ciertas reglas específicas, con esto se crea un árbol denominado AST, tal como fue mencionado en el capítulo 5. Parte del analizador sintáctico se muestra en la Figura 19.

```
model:         (variableSet)? (constraintSet)?
               {## = #( #[MODEL, "MODEL"] ,##);};

variableSet:   RES_VARIABLES! colon (variable)*
               { ## = #( #[VARIABLE_SEQ, "VARIABLE_SEQ"] ,##);};

constraintSet: RES_CONSTRAINTS! colon (constraint)*
               { ## = #( #[CONSTRAINT_SET, "CONSTRAINT_SET"] ,##);};
```

Figura 19: Porción del analizador sintáctico en RSSolver.

El árbol o AST proporcionado por el analizador sintáctico o *parser*, se utiliza en la siguiente etapa de compilación, el análisis semántico. En esta parte del proceso se recorre el modelo a través del árbol generado (rsSolverTreeParser.g) e interactúa con las clases utilizadas para la inspección semántica (rsSolver.compilers.inspector), tal como se muestra en la Figura 20.

```
decVar:        #(DEC_VAR (idVar:IDENT #(INTERVAL (min:LIT_INT max:LIT_INT)))
               {ml.addVar(idVar.getText(),min.getText(),max.getText());} ) ;

constraintSet: #(CONSTRAINT_SET (constraint)*);

constraint:    #(c:CONSTRAINT {ml.addConstraint(c.getFirstChild());});

expression:    #(rEqv:OP_EQV    expression expression)
               {qB.addOp(rEqv);}
```

Figura 20: Porción del analizador semántico en RSSolver.

En el código expuesto se pueden ver elementos como mI o qB, que corresponden a las clases ModelInspector y QueueBuilder respectivamente, las que en conjunto con CommonMethodsManager permiten finalizar la etapa de compilación. Los últimos elementos mencionados son expuestos de forma genérica a continuación:

- **CommonMethodsManager**: Posee dos métodos, isOperator e isInteger. El primero de ellos retorna true si es un operador y está dentro

de los operadores permitidos. El segundo método es utilizado de la misma forma que el primero pero asociado a los números enteros.

```
1   public boolean isOperator(String op) {
2       if(op.equals("<->")   || op.equals("<-")  ||
3          op.equals("->")    || op.equals("or")  ||
4          op.equals("xor")   || op.equals("and") ||
5          op.equals("equal") || op.equals("<>")  ||
6          op.equals("!=")    || op.equals("<")   ||
7          op.equals(">")     || op.equals("<=")  ||
8          op.equals(">=")    || op.equals("=<")  ||
9          op.equals("=>")    || op.equals("+")   ||
10         op.equals("-")     || op.equals("*")   ||
11         op.equals("/" ))
12         return true;
13      else
14         return false;
15  }
16
17  public boolean isInteger(String op) {
18      try{
19          Integer.parseInt(op);
20          return true;
21      } catch(Exception e) {
22          return false;
23      }
24  }
```

Figura 21: Métodos isOperator y isInteger en RSSolver.

- **ModelInspector**: Mediante los métodos addVar y addConstraint se añaden las variables con sus respectivos dominios, además de las restricciones al modelo que se utilizará para trabajar con el Solver.

```
1   public void addVar(String id, String min, String max) {
2       Tool.getMODEL().addVar(id,min,max);
3   }
4
5   public void addConstraint(AST ast) {
6       Tool.getMODEL().addConstraint(ast);
7   }
```

Figura 22: Métodos addVar y addConstraint en RSSolver.

- **QueueBuilder**: Mediante los métodos de la clase CommonMethodsManager se crea una cola que contiene los operadores y operandos, además de mostrar un mensaje de error cuando sea necesario.

Todo lo mencionado en este subcapítulo hace referencia al proceso que comienza en el momento en que se rescata el modelo, hasta que queda todo dispuesto para comenzar la búsqueda. Todo este procesamiento se hace con el objetivo de traspasar cada uno de los elementos del modelo a objetos, los cuales se pueden trabajar directamente con el lenguaje Java.

### 7.2.2. Comprensión y funcionamiento

Existen tres clases importantes para entender el funcionamiento tras el proceso de compilación (ver Figura 24 y Figura 25), la primera de ellas es la clase padre denominada *MODEL*, que contiene tres estructuras de datos como atributos, las cuales son:

- **variables**: Almacena las variables y sus respectivos valores en un momento determinado. Corresponde a un *HashMap*, por lo tanto cada variable corresponde a un objeto de tipo *Variable* con un identificador para cada uno.
- **orderedVars**: Contiene una lista de los nombres de cada variable existente en el modelo.
- **constraints**: Corresponde a una lista de objetos de tipo *Constraint*, es decir, las restricciones asociadas al modelo. Estas se almacenan de la siguiente forma:

$$[[x = null, y = null, +, 10, >=], [x = null, y = null, +, 10, <=]]$$

Figura 23: Restricción en un array en RSSolver.

Esto corresponde al ejemplo presentado en la Figura 15, pero se debe tomar en cuenta que los valores de los dominios de las variables irán reemplazando a *null* en la medida que sea necesario.

La segunda clase recibe el nombre *VARIABLES* y contiene 4 atributos:

- **id**: Para cada variable del modelo se tiene un identificador, que corresponde al nombre de esta almacenado como String.

- **minBound**: Del modelo se extrae la cota inferior del dominio de cada variable y se almacena como un número entero.
- **maxBound**: Del modelo se extrae la cota superior del dominio de cada variable y se almacena como un número entero.
- **value**: Este atributo va a contener el valor asociado a una variable en un momento determinado.

Finalmente se tiene una clase con el nombre *CONSTRAINT*, ésta contiene un *ArrayList* con el nombre **expression**, éste va almacenar las restricciones de la misma forma que se muestra en la clase MODEL.

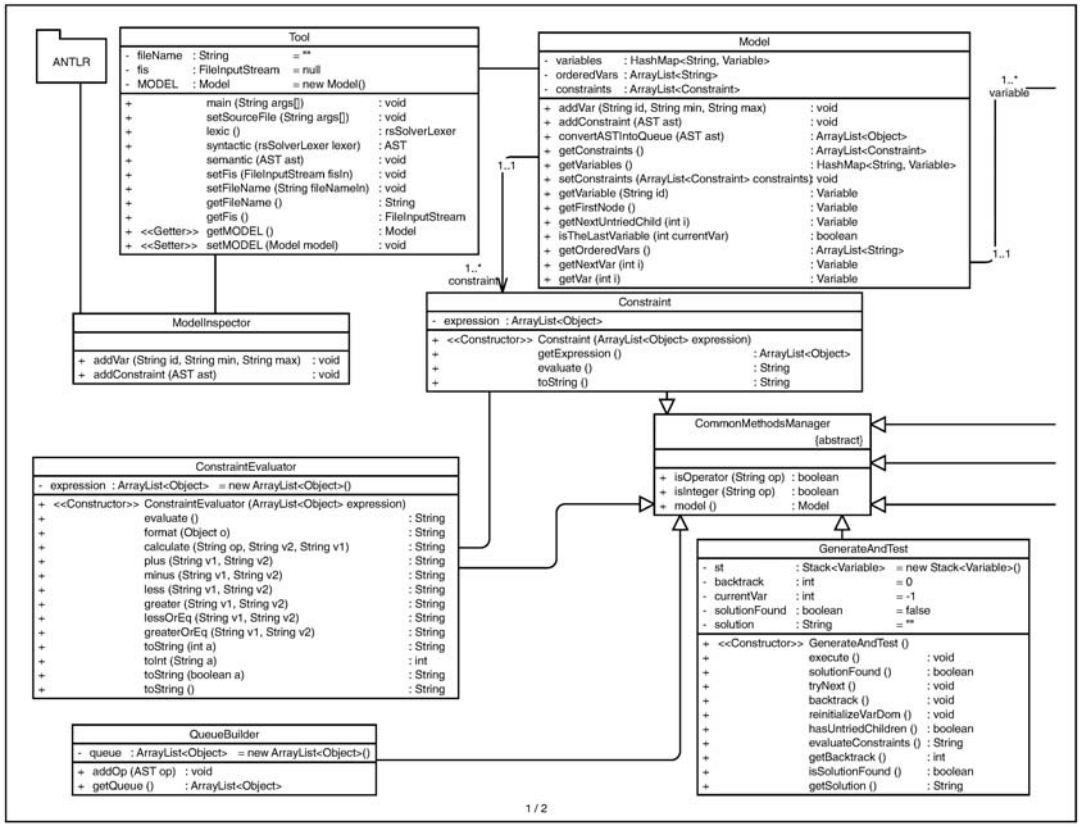



Figura 24: Diagrama de clases RSSolver, parte 1.

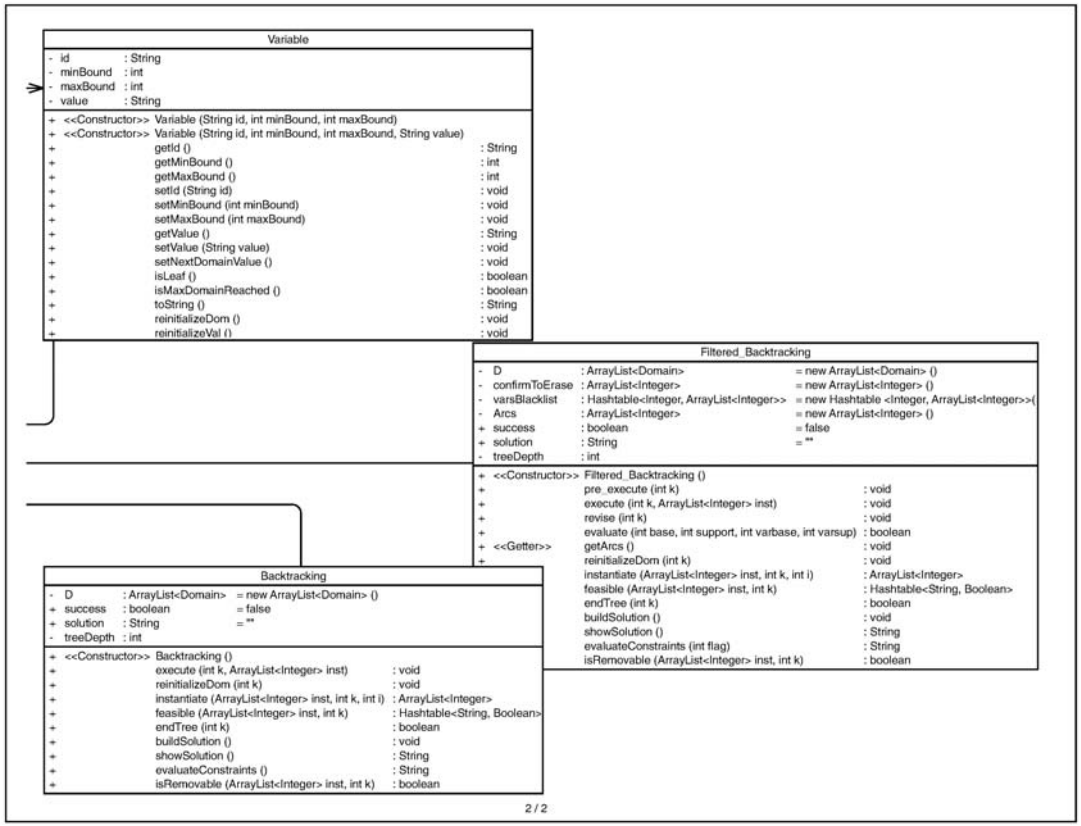



Figura 25: Diagrama de clases RSSolver, parte 2.

## 7.3. Incorporación de algoritmo de consistencia a RSSolver

Una vez comprendido el funcionamiento completo del Solver, es posible trabajar sobre él pensando en el objetivo del proyecto. Como se ha mencionado en los tópicos 3.2 y 4, existen diversos algoritmos de búsqueda, a los cuales es posible aplicar distintos tipos de arco-consistencias, éstos actúan como un filtro sobre los dominios de las variables, filtro que permite que la búsqueda realizada para encontrar la solución sea más eficiente. En el RSSolver se ha trabajado en dos algoritmos de búsqueda (Generate and Test y Backtracking), a través de los cuales se obtiene una solución, que no necesariamente es óptima, pero que cumple claramente con las restricciones de cada modelo. Debido a la necesidad de mejorar el Solver con el que se trabaja, se ha decidido realizar un filtrado, pensando en AC-1 en conjunto con Backtracking, de tal manera de lograr un *CSP Mínimo Equivalente*.

Debido a que existe la posibilidad de modelar todo tipo de problemas en función a restricciones binarias, se ha decidido trabajar solo con arcos. La incorporación del filtraje se ha realizado como una etapa previa al proceso de búsqueda de solución, pero se debe considerar que para realizar el filtraje, de todas formas se deben evaluar las restricciones previamente, lo cual hace pensar que el filtrado en problemas pequeños no se notará o simplemente tendrá un resultado inverso al esperado.

Para realizar el filtraje se ha considerado agregar ciertos elementos al Solver, los cuales son especificados a continuación:

- Se agregaron 3 estructuras de datos:

1. $ArrayList < Integer >$ llamado $Arcs$.
2. $ArrayList < Integer >$ llamado $reverse\_Arcs$.
3. $ArrayList < Integer >$ llamado $toErase$.

- A través del método **pre-execute**, primero se obtienen los arcos que estén involucrados en las restricciones por medio del método *getArcs* (*línea 2*), después de lo anterior se obtienen los arcos inversos para lograr realizar un análisis bidireccional(*línea 3*), a continuación se determinan los valores inconsistentes de los dominios de las variables (mediante los métodos *revise* y su submétodo*evaluate*) (*línea 7*). A continuación se inicializan las variables a su estado inicial, ya que al buscar inconsistencias, éstas quedan con asignaciones y para realizar la búsqueda en Backtracking deben tener valor *null* (*línea 8 y línea 13*), todo lo anterior va dentro de un *Do-While*(*línea 4 a 16*), que está

condicionado por *exitPreExecute*, ya que de esta manera se logra la iteratividad necesaria para llegar al CSP mínimo equivalente, realizando todos los ciclos necesarios hasta que no se encuentren mas inconsistencias. Luego se ejecuta el método *execute*, el cual recibe el arreglo con los valores inconsistentes (*línea 17*).

```
1  public void pre_execute(int k){
2    getArcs();
3    revertArcs();/*Invierte los arcos para lograr la bidereccionalidad↩
       */
4    do{
5      exitPreExecute=0;
6      chosenConstraint=0;
7      revise(k,Arcs);
8      for(int i=0;i<D.size()-1;i++){
9          model().getVar(i).setValue(null);
10     }
11     chosenConstraint=0;
12     revise(k,reverse_Arcs);
13     for(int i=0;i<D.size()-1;i++){
14         model().getVar(i).setValue(null);
15     }
16   }while(exitPreExecute!=0){
17         execute(0, new ArrayList<Integer>());
18     }
19 }
```

Figura 26: Método pre-execute en RSSolver.

- Pre-execute ejecuta el método **getArcs**, este método determina cuales son los arcos que están involucrados en las restricciones, es decir, si una restricción es *x+y<500*, el arco involucrado sería *(x,y)*, donde el primer *for* recorre las restricciones contenidas en el modelo (*línea 2*), despues se utiliza un *for* sólo para determinar las 2 variables involucradas, ya que sólo se trabajan con restricciones binarias(*línea 3*). A continuación se emplea otro *for*, que recorre cada variable existente en el modelo(*línea 4*) y finalmente un *if* que tiene la función de determinar si en la restriccion existe o no la variable que se necesita(*línea 5*), si la respuesta es *true*, lo guardara en el ArrayList Arcs (*línea 6*) he inmediatamente se emplea *break* para evitar búsquedas innecesarias (*línea 7*).

```
1  public void getArcs(){
2      for(Constraint con:model().getConstraints()){
3          for(int i=0;i<2;i++){
4              for(int j=0;j<model().getOrderedVars().size();j++){
5                  if(con.getExpression().get(i).toString().contains(↩
                       model().getOrderedVars().get(j).toString())){
6                      Arcs.add(j);
7                      break;
8                  }
9              }
10         }
11     }
12 }
```

Figura 27: Método getArcs en RSSolver.

- Pre-execute ejecuta el método **revertArcs**, este método se encarga de obtener los arcos inversos, necesarios para lograr una búsqueda bidireccional de inconsistencias. Para lograrlo se utilizan dos variables locales, *sizeArcs* y *base* (*línea 2 y 3*), que ayudarán a recorrer el arreglo donde están contenidos lo arcos directos, estos se recorren mediante el *for* de la *línea 4* y dentro de ésta se procede a guardar los arcos inversos en el array *reverse_ Arcs* en la *línea 6 y 7*.

```
1  public void revertArcs(){
2      int sizeArcs = Arcs.size();
3      int base=0;
4      for(int i=0;i<(sizeArcs/2);i++)
5      {
6      reverse_Arcs.add(base, Arcs.get(base+1));
7      reverse_Arcs.add((base+1), Arcs.get(base));
8      base=base+2;
9    }
10 }
```

Figura 28: Método revertArcs en RSSolver.

- Pre-execute ejecuta el método **revise**, este método toma cada uno de los elementos del dominio de la base y sus posibles soportes (*línea 4 - 5*), después cada arco que se obtuvo de la función *getArc*, es evaluado para saber si el valor de la base puede ser considerado como solución.

Para determinar la consistencia o inconsistencia de la base, se ejecuta el método evaluate (*línea 6*), en el caso que el valor sea inconsistente es almacenado en la lista Toerase (*línea 10*) y a la variable global *exitPreExecute* incrementa en uno, esto para determinar que se encontraron inconsistencias y por ende se debe iterar una vez más sobre los arcos(directos e inversos) en la búsqueda de inconsistencias hasta que exitPreExecute no cambie más de valor, a continuación se determina si los valores contenidos en Toerase pertenecen al tramo inicial (*línea 16*) o final (*línea 19*) del dominio de la base, dependiendo del caso, se modifican los dominios directamente del modelo (*línea 18 y línea 20*). Después se inicializan los valores de las variables en null para no tener problemas de asignaciones que se puedan generar en una futura búsqueda(*línea 23*) y finalmente se cambia el valor de la variable *chosenConstraint* (*línea 26*), para realizar la evaluación sobre una restricción en especifico y a continuación se hace un llamado recursivo de la función *revise*(*línea 27*).

```
1  public void revise(int k){
2      ArrayList<Integer> Toerase = new ArrayList<Integer> ();
3      boolean Ev = false;
4      for (int base= D.get(Arcs.get(k)).minBound; base<=D.get(Arcs.get↩
           (k)).maxBound; base++){
5          for (int support= D.get(Arcs.get(k+1)).minBound; support<=D.↩
               get(Arcs.get(k+1)).maxBound; support++){
6              Ev = (evaluate(base, support, Arcs.get(k), Arcs.get(k+1)↩
                   )||Ev);
7              if(Ev) break;
8          } if(!Ev){
9              Toerase.add(base);
10             exitPreExecute++;
11         }
12         Ev=false;
13     } if(!Toerase.isEmpty()){
14         if(Toerase.get(0)== D.get(Arcs.get(k)).minBound){
15             D.get(Arcs.get(k)).minBound= (Toerase.get((Toerase.size↩
                   ())-1))+1;
16         } else{
17             D.get(Arcs.get(k)).maxBound= (Toerase.get(0))-1;
18         }
19     } for(int i=0;i<D.size()-1;i++){
20         model().getVar(i).setValue(null);
21     } chosenConstraint=chosenConstraint+1;
22     if((k+1)!=(Arcs_evaluation.size()-1)) revise((k+2),↩
           Arcs_evaluation);
23     model().getVar(D.size()-1).reinitializeDom();
24     }
```

Figura 29: Método revise en RSSolver.

- **Evaluate** específicamente se encarga de evaluar las restricciones, considerando una base evaluada con cada valor del soporte. Si la base considerada posee al menos un soporte, evaluate retorna inmediatamente true, esto quiere decir que la base que se está evaluando puede ser considerada dentro de una solución, el caso contrario ocurre cuando el dominio es evaluado con todos los posibles soportes sin tener éxito, en este caso el valor retornado será false, lo que indica que la base actual no puede ser parte de la solución, por lo que puede ser eliminada. Para lograr lo anteriormente mencionado, recibe por parámetro los valores de la base y soporte, además de sus respectivas variables (*línea 1*), después se realiza la asignación correspondiente a cada variable con sus respectivos valores (*línea 3 - 4*), a continuación se evalúan con la función *preEvaluateConstraints*(*línea 5*), que es similar a la función

*evaluateConstraints* pero con la diferencia de que sólo realiza la evaluación sobre una restricción determinada, y según sea la respuesta, se retorna false o true (*línea 7 y línea 8*).

```
1  public boolean evaluate(int base, int support, int varbase, int ↩
       varsup){
2      String evaluated;
3      model().getVar(varbase).setValue(Integer.toString(base));
4      model().getVar(varsup).setValue(Integer.toString(support));
5      evaluated = preEvaluateConstraints(0);
6      if((evaluated!="goal")&&(evaluated!="insufficient-instantiation"↩
           )){
7          return false;
8      }else return true;
9  }
```

Figura 30: Método evaluate en RSSolver.

- **Execute**, se encarga de crear el árbol de soluciones, el cual se recorre mediante el algoritmo Backtracking para encontrar una solución. Execute recibe el resultado del proceso de filtraje. Proceso que reduce, dependiendo del modelo, el árbol de búsqueda de Backtracking, logrando evitar la instanciación de los elementos inconsistentes del modelo.

```
1  public void execute(int k, ArrayList<Integer> inst) {
2      for (int i= D.get(k).minBound; i<=D.get(k).maxBound && !success;↩
           i++) {
3          inst = instantiate(inst,k,i);
4        System.out.println("evaluation: " + showSolution());
5        Hashtable<String,Boolean> h = feasible(inst,k);
6        if(h.get("cons")) {
7          if(h.get("solution-found")){
8              buildSolution();
9            success=true;
10         }else
11             execute(k+1, inst);
12         }
13   reinitializeDom(k);
14 }
```

Figura 31: Método execute en RSSolver.

# 8. Experimentos

Considerando que los detalles entregados en los capítulos anteriores, sobre los grandes resultados obtenidos a partir del algoritmo de consistencia propuesto, no bastan para demostrar la eficiencia del mismo, se presentará una serie de experimentos, o más bien un paralelo entre las distintas posibilidades que se barajan en el RSSolver y que a su vez son comparables entre sí, demostrando de esta forma los reales resultados observables con pruebas concretas basadas en el tiempo observado, para llegar al resultado del problema en cuestión.

## 8.1. El Problema

La elección del problema es un punto importante a la hora de realizar pruebas, específicamente lo que respecta al dominio, debido a que la elección de un problema con dominios pequeños no muestra resultados representativos. Todo esto se debe a que la mejora del algoritmo se basa en el tiempo que se ahorra al eliminar los valores inconsistentes de los dominios de las variables involucradas en un problema específico. Al considerar dominios pequeños, existen menos posibilidades de eliminar un gran número de valores del dominio de alguna variable, lo que se traducirá finalmente en que el problema no mejore su tiempo de respuesta.

Por todo lo mencionado anteriormente, el problema a utilizar será el que se presenta a continuación, ya que los dominios y las restricciones son un buen caso de prueba para medir la eficacia del algoritmo.

$$
\begin{array}{c}
variables: \\
\hline
x\ in\ [1, 1000]; \\
y\ in\ [12, 53]; \\
z\ in\ [1, 45]; \\
\\
constraints: \\
\hline
x + y > 750; \\
y + z > 60; \\
x - z > 860;
\end{array}
$$

## 8.2. Resultados

La idea principal de este capítulo es mostrar a través de hechos concretos que la aplicación del algoritmo de consistencia supera al algoritmo puro,

incluso superando el tiempo de búsqueda total. Esto es muy importante, ya que, se debe tener en cuenta que AC1 actúa como etapa de pre proceso al algoritmo de búsqueda, ejecutándose, en algunos casos, una gran cantidad de veces, debido a que busca llegar al CSP mínimo equivalente, revisando el árbol de búsqueda tantas veces como sea necesario, posteriormente se aplica al CSP mínimo equivalente el algoritmo de Backtracking, llegando de esta forma el resultado que satisface todas las restricciones. A continuación se presenta una tabla con los tiempos de búsqueda de dos modelos, el primero de ellos es el caso presentado en el punto 1.1, el cual es un buen caso de prueba. El segundo será un modelo seleccionado al azar en el momento de realizar las pruebas, el cual de todas formas seguirá las bases presentadas en el punto anterior, pero no será un caso estudiado con anterioridad.

**Modelo nº1**

$variables$ :

---

$x\ in\ [1, 1000]$;
$y\ in\ [12, 53]$;
$z\ in\ [1, 45]$;

$constraints$ :

---

$x + y > 750$;
$y + z > 60$;
$x - z > 860$;

**Modelo nº2**

$variables$ :

---

$x\ in\ [70, 1000]$;
$y\ in\ [7, 2000]$;
$z\ in\ [1, 275]$;

$constraints$ :

---

$x + y > 1050$;
$y + z > 1120$;
$x - z > 860$;

En la Tabla 5 se muestran los dos casos de prueba utilizados, para mostrar el comportamiento del algoritmo puro en comparación al algoritmo que

| Modelo | Promedio Backtracking puro | Promedio Backtracking+AC1 mejorado |
|---|---|---|
| Nº1 | 831 ms. | 350 ms. |
| Nº2 | 511554 ms. | 1120 ms. |

Tabla 5: Comparación de tiempos entre Backtracking puro y Backtracking con AC-1 mejorado.

utiliza filtraje. Como se puede apreciar en el modelo nº1, Backtracking puro llega a la solución en 831 ms., Backtracking + AC1 mejorado por su parte, resuelve el problema en 350 ms., mostrando una mejoría de un 57,8 %.

En el modelo nº2 se decidió utilizar números más elevados, tanto para los dominios, como para las restricciones, lo cual no había sido probado con anterioridad. Finalmente los resultados fueron los esperados, considerando que Backtracking puro demoró 511554 ms. en encontrar el resultado, mientras que Backtracking + AC1 mejorado solamente demoró 1120 ms., lo cual corresponde a una notable mejoría del 99,78 %.

Al finalizar los experimentos, no solo es posible mencionar que el algoritmo de filtrado funciona, sino que además se puede afirmar que entre más amplios sean los dominios, mayor es la mejoría mostrada por parte del algoritmo.

# 9. Conclusión

La investigación realizada, muestra claramente que la programación con restricciones y específicamente los CSP son un importante recurso para enfrentar un sinnúmero de problemas, pero no cualquier problema puede ser expresado como un CSP, debido a que se tiene que cumplir con las condiciones para lograr un modelo: identificación de variables, dominios asociados a cada una de ellas y un número finito de restricciones entre una o más variables. A través de la información recopilada y analizada, es destacable que los elementos asociados a este tipo de problemas no trabajan de forma individual, sino que cada nuevo elemento es un complemento para el anterior, así cuando se menciona el concepto de algoritmo de búsqueda, indistintamente de cual sea, puede ser asociado a los diferentes tipos de técnicas de inferencia o consistencia, ya sea de nodo, arco, camino u otra, ayudando a reducir los espacios de búsqueda y mejorando notablemente la eficiencia en el trabajo de resolución, de la misma manera las técnicas de enumeración, siempre y cuando sean seleccionadas correctamente, ayudan y se complementan con los elementos anteriormente mencionados.

La misma idea se cumple en relación al tema específico de los algoritmos de arco consistencia, en donde, a través del tiempo, las falencias de uno ayudan a mejorar las futuras ideas, desarrollando técnicas cada vez más eficientes, apoyándose siempre en uno o más algoritmos pasados.

En la etapa de implementación a quedado en claro que entre la teoría y la puesta en práctica existe un gran trecho, lleno de dificultades, como el hecho de lograr comprender el funcionamiento interno de un Solver, específicamente RSSolver, construido bajo un lenguaje Orientado a Objetos, el cuestionamiento de cómo lograr crear un algoritmo de consistencia o filtrado que provocara el menor grado de modificación al código original, aprovechando al máximo las características contenidas en el mismo, y finalmente, cómo lograr que el algoritmo de Backtracking no tome en cuenta aquellos valores que nunca formarían parte de una solución. Todos estos problemas se fueron solucionando gracias a un exhaustivo trabajo grupal, teniendo como resultado un algoritmo capaz de optimizar Backtracking, resultando una búsqueda en menor tiempo mediante un código bastante simple, que dé como resultado un CSP mínimo equivalente.

Considerando el factor tiempo y a las dificultades para comprender el problema y el funcionamiento de la herramienta con la que se trabajó, es posible mencionar que se ha cumplido con las expectativas esperadas, ya que a través de ésta experiencia se ha logrado comprender con mayor detalle el funcionamiento de RSSolver, logrando en su totalidad el cumplimiento de los objetivos propuestos.

# 10. Referencias

## Referencias


[1] Marlene Arangú and Miguel Salido. A fine-grained arc-consistency algorithm for non-normalized constraint satisfaction problems. *Int. J. Appl. Math. Comput. Sci.*, 21(4):733–744, December 2011.

[2] Marlene Arangú, Miguel Salido, and Federico Barber. Ac4-op: optimizing a fine-grained arc-consistency algorithm. In *Conferencia de la Asociación Española para la Inteligencia Artificial y Jornadas de Transferencia Tecnológica de Inteligencia Artificial*, 2009.

[3] Marlene Arangú, Miguel A. Salido, and Federico Barber. Ac3-op: An arc-consistency algorithm for arithmetic constraints. In *Proceedings of the 2009 conference on Artificial Intelligence Research and Development: Proceedings of the 12th International Conference of the Catalan Association for Artificial Intelligence*, pages 293–300, Amsterdam, The Netherlands, The Netherlands, 2009. IOS Press.

[4] Marlene Arangú, Miguel A. Salido, and Federico Barber. Extending arc-consistency algorithms for non-normalized csps. In *SGAI Conf.'09*, pages 311–316, 2009.

[5] Federico Barber and Miguel A. Salido. Introducción a la programación de restricciones. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 20:13–30, 2003.

[6] Roman Bartak, Miguel A. Salido, and Francesca Rossi. New trends in constraint satisfaction, planning, and scheduling: A survey. *Knowledge Eng. Review*, 25:249–279, 2004.

[7] Christian Bessiere, Thierry Petit, and Bruno Zanuttini. Making bound consistency as effective as arc consistency. In *IJCAI*, 2009.

[8] Enrique José García Cota. Guía práctica de antlr 2.7.2, versión 1.0. Universidad de Sevilla. Departamento de Lenguajes y Sistemas Informáticos, Septiembre 2003.

[9] Broderick Crawford, Carlos Castro, and Eric Monfroy. Programación con restricciones dinámica. Technical report, Pontificia Universidad Católica de Valparaíso, Universidad Técnica Santa María, Université de Nantes, 2009.

[10] Robert Haralick and Gordon Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[11] Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse grained arc consistency algorithms. In *In Proceedings of CP 2003*, pages 480–494, 2003.

[12] Marlene Alicia Arangú Lobig and Miguel Ángel Salido Gregorio. *Modelos y Técnicas en Problemas de Satisfacción con Restricciones*. PhD thesis, Universidad Politécnica de Valencia, 2011.

[13] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[14] Felip Manyà and Carla Gomes. Técnicas de resolución de problemas de satisfacción de restricciones. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 7:169–180, 2003.

[15] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[16] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[17] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[18] Roque Marin Morales. *Inteligencia Artificial: Técnicas, métodos y aplicaciones*. McGraw-Hill, 2008.

[19] Irene Bárbara Rodríguez and Carmelo del Valle Sevillano. *Algoritmos de planificación basados en restricciones para la sustitución de componentes defectuosos*. PhD thesis, Universidad de Sevilla, 2007.

[20] David L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical report, Massachusetts Institute of Technology, 1972.